Ionic 4

Desenvolvimento multiplataforma para dispositivos móveis



Casa do Código

EVERTON COIMBRA DE ARAÚJO

Sumário

- ISBN
- Agradecimentos
- Sobre o autor
- Prefácio
- Sobre o livro
- 1. Dispositivos móveis, desenvolvimento cross-platform e Ionic
- 2. Ionic — Instalação e testes
- 3. A primeira aplicação Ionic
- 4. Listagem de dados e navegação entre páginas
- 5. Persistindo dados fisicamente em aplicações Ionic
- 6. Side Menu, CRUD com SQLite e o componente Select
- 7. CRUD com Firestore para acesso a dados remoto e uma página de pesquisa
- 8. Câmera e álbum de fotos no CRUD de clientes e Storage do Firebase
- 9. Autenticação e criação de usuários no Firebase e uso de Tabs
- 10. Integração com Google Maps
- 11. Os estudos não param por aqui

ISBN

Impresso e PDF: 978-65-86110-13-5

EPUB: 978-65-86110-12-8

MOBI: 978-65-86110-14-2

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Agradecimentos

Este é um espaço que considero muito importante em meus livros. É o momento de reconhecimento ao apoio direto ou indireto de pessoas importantes para mim, durante o processo de pesquisa e escrita do livro.

Não posso jamais deixar de agradecer ao apoio de meus filhos, pessoas chaves em minha vida. No momento em que me encontro, pessoal e profissionalmente, duas pessoinhas são muito importantes para os momentos de relaxamento: Maria Clara e Vicente Dirceu. Meus caçulas, minhas joias mais preciosas. Meu filho Gabriel e minha nora Camila, conversamos sempre à noite, em momentos em que me divirto muito com eles. Comecei a escrever este livro, também, na casa deles. Obrigado, meus filhos.

Em relação à revisão técnica, uma vez mais meu sincero e grande agradecimento à excelente pessoa, competentíssimo profissional: Leonan Fraga Leonardo, que mantém sempre seu LinkedIn atualizado. Hoje o Leonan está de volta ao Brasil, e muito bem colocado em uma grande multinacional de TI, focado em aplicações web e para dispositivos móveis, e sempre está disponível para troca de informações.

Terminando a escrita do livro, comecei a pensar em quem convidar para o prefácio. De imediato me veio a imagem de uma pessoa de muito bom coração e excelente profissional. Falo do César Angonese, que, quando entrei na UTFPR, me recebeu de braços abertos, me auxiliando em momentos difíceis e sempre oportunizando situações.

Embora este texto seja repetitivo, sua força só aumenta à cada vez que o faço. Agradeço muito à Casa do Código, principalmente à Vivian Matsui, um anjo que sempre me acompanha nos trabalhos que desenvolvo, sempre paciente e procurando tempo para ler e contribuir com as revisões dos livros. Obrigado, Vivian e este agradecimento se estende a toda a equipe da editora.

De maneira geral, agradeço sempre aos meus colegas de trabalho do Departamento de Computação da UTFPR, câmpus Medianeira, meu paraíso profissional, pelo coleguismo e pelos ouvidos.

Sobre o autor



Everton Coimbra de Araújo atua na área de ensino de linguagens de programação e desenvolvimento. É tecnólogo em processamento de dados pelo Centro de Ensino Superior de Foz do Iguaçu, possui mestrado em Ciência da Computação pela UFSC e doutorado pela UNIOESTE em Engenharia Agrícola, na área de Estatística Espacial. É professor da Universidade Tecnológica Federal do Paraná (UTFPR), câmpus Medianeira, onde leciona disciplinas no Curso de Ciência da Computação e em mestrados. Já ministrou aulas de Algoritmos, Técnicas de Programação, Estrutura de Dados, Linguagens de Programação, Orientação a Objetos, Análise de Sistemas, UML, Java para Web, Java EE, Banco de Dados e .NET.

Possui experiência na área de Ciência da Computação, com ênfase em Análise e Desenvolvimento de Sistemas, atuando principalmente

nos seguintes temas: Desenvolvimento Web com Java e .NET, Persistência de Objetos e agora em Dispositivos Móveis. O autor é palestrante em seminários de informática voltados para o meio acadêmico e empresarial.

Prefácio

O professor Dr. Everton Coimbra de Araújo já escreveu diversos livros na área de informática, que são utilizados em diversas universidades como referência nas disciplinas. Eu mesmo já utilizei alguns como guia para ministrar minhas aulas na UTFPR câmpus Medianeira, sempre com bons resultados.

O diferencial nos livros do Dr. Everton é que ele tem conhecimento didático sobre os assuntos que escreve e sempre fazendo referência com o dia a dia nas empresas, o que torna a obra muito mais interessante.

O assunto abordado neste livro é realmente o que profissionais da área de desenvolvimento estavam precisando para desenvolver seus aplicativos para dispositivos móveis.

Há pouco tempo, quando se decidia desenvolver aplicativos, deveria se optar para o sistema operacional Android ou iOS, sendo despendido muito esforço para adaptar para os dois sistemas operacionais. Com o framework apresentado neste livro, o Ionic, o desenvolvedor deverá dispender esforços apenas na sua aplicação, sendo transparente a questão do sistema operacional.

Lendo este livro me resta pouca saudade do desenvolvimento de aplicativos para Palmtop, PDAs ou HandHelds. Como ficou fácil desenvolver aplicativos para os usuários trabalharem remotamente sem necessidade de conectar através dos modens ou levar o equipamento até a empresa para transferência dos dados.

O autor foi muito feliz na forma como escreveu este livro. Nos capítulos iniciais, faz uma contextualização de dispositivos móveis e ferramentas utilizadas, de forma que o leitor não precisa ser um expert. Em seguida, é apresentada a instalação e teste da plataforma que será utilizada. Ele vai explicando o assunto e ligando a leitura aos conteúdos dos capítulos seguintes com uma forma fácil, didática e interessante de aprendizado.

No final da leitura deste livro você estará apto para desenvolver bons aplicativos comerciais, utilizando uma das melhores ferramentas disponíveis no mercado para dispositivos móveis. Uma boa leitura.

César Angonese

Universidade Tecnológica Federal do Paraná - Câmpus Medianeira

Sobre o livro

Este livro traz, na prática, o desenvolvimento de aplicações *cross-platform* com o Ionic, um framework para desenvolvimento de aplicativos para dispositivos móveis e web. Desenvolver um aplicativo para ser publicado em dispositivos com plataformas diferentes (iOS e Android) é uma tarefa tranquila com o Ionic. É possível criar uma aplicação, utilizando a linguagem HTML e TypeScript, e ela ser publicada para as duas plataformas, além do navegador.

O livro é desenvolvido em dez capítulos. O primeiro é apenas teórico, mas não menos importante, pois trago nele contextualizações sobre dispositivos móveis e as ferramentas usadas no livro. Já no segundo, apresento o processo de instalação e teste da plataforma e do IDE que utilizaremos, o Visual Studio Code. Publicaremos aplicações no navegador e no Electron, um framework para que aplicações JavaScript possam ser executadas no desktop.

A aplicação a ser implementada durante o livro refere-se a uma oficina mecânica, com cadastro de Clientes, Serviços, Peças e Atendimentos. Persistiremos os dados localmente, no navegador, depois em uma base de dados local e, ao final, fazendo uso de recursos do Firebase. Ofereceremos ao usuário a possibilidade de capturar fotos no momento do cadastro do cliente.

A prática começa bem legal no capítulo 3, onde criaremos interfaces com o usuário para interação com a aplicação. Veremos alguns recursos possíveis para tratamento dos dados informados, validando e informando ao usuário possíveis inconsistências. Veremos nossos primeiros controles visuais e buscaremos entender a estrutura de aplicações Ionic.

No quarto capítulo, trabalharemos a navegação entre páginas, criaremos uma página para listagem de dados e a partir desta página possibilitaremos a navegação para outra. Também teremos a

introdução em serviços na concepção de componentes do Ionic. Teremos neste capítulo um CRUD completo, onde poderemos inserir, recuperar, atualizar ou remover dados.

No capítulo 5, trabalharemos a persistência física de dados, iniciando com o Storage do Ionic, que mantém os dados no navegador, onde a aplicação é executada. Após este conhecimento, faremos uso de um banco de dados, o SQLite, também para a persistência física dos dados. Para os dois cenários teremos as operações CRUD implementadas.

No capítulo 6, traremos o conceito de Sidemenu do Ionic, que é uma página que serve como um menu de opções disponibilizadas pela aplicação para o usuário. Começaremos aqui o trabalho de associação de classes em nossas páginas de interação com o usuário.

Dando sequência ao tema de persistência, no capítulo 7, conheceremos o Firebase e vamos utilizar o Firestore, que é uma solução do Google para persistência e sincronização de dados em aplicações web e para dispositivos móveis. Um capítulo muito legal.

No capítulo 8 interagiremos com a câmera e álbum de fotos de nossos dispositivos físicos. Possibilitaremos ao usuário capturar uma foto de um cliente ou selecioná-la de seu álbum de fotografias. Inicialmente gravaremos esta imagem em nosso dispositivo. Depois, no mesmo capítulo, faremos uso de outra ferramenta da Google, o Storage, que é capaz de armazenar imagens na nuvem. Veremos as operações CRUD também para estas imagens e as associaremos com nossos clientes.

Já nos aproximando do final do livro, veremos uma nova maneira de navegação entre páginas oferecidas pelo Ionic, as guias (Tabs). Para ilustrar seu uso, trabalharemos com autenticação do usuário e, uma vez, mais, recorrendo ao Firebase, utilizaremos a ferramenta Authentication e você verá que é muito simples e flexível.

Enfim, como último capítulo, o de número 10 traz a integração com o Google Maps. Pesquisaremos endereços e buscaremos nossa localização. Fechando o capítulo e o livro, veremos como alterar a imagem de inicialização de nossa aplicação e o ícone que ela terá em nosso dispositivo.

Certamente, este livro pode ser usado como ferramenta em disciplinas que trabalham o desenvolvimento de dispositivos móveis, quer seja por acadêmicos ou professores. Isso porque ele é o resultado da experiência que tenho em ministrar aulas dessa disciplina, então trago para cá anseios e dúvidas dos alunos que estudam comigo. Eu já utilizo este conteúdo com meus alunos e eles gostam muito.

É importante que o leitor tenha conhecimento de Orientação a Objetos e de alguma linguagem de programação, mas não é um fator impeditivo. Conhecimentos básicos sobre banco de dados também são interessantes. O repositório com todos os códigos-fontes usados no livro pode ser encontrado em: https://github.com/evertonfoz/implementacoes-de-livros/tree/master/ionic.

Os arquivos disponibilizados no GitHub estão de acordo com as versões apontadas no livro. Recomendo que, inicialmente, você implemente os exemplos nestas versões e, após o sucesso, tente atualizar para a mais recente, já que é possível que haja essa atualização quando esteja lendo o livro, uma vez que a equipe do Ionic é muito dinâmica e sempre está disponibilizando atualizações evolutivas e corretivas. Reforçando, fique atento ao caso de atualizar os plugins e componentes dos projetos, pois essa tecnologia é dinâmica e atualizações estão sempre ocorrendo. Coloco-me sempre à disposição para estes casos, via email direto, evertoncoimbra@gmail.com.

Que a leitura deste livro seja para você tão prazerosa como para mim foi escrevê-lo. Desfrute, sem moderação. Sucesso.

Capítulo 1
Dispositivos móveis, desenvolvimento cross-platform e Ionic

Olá! Seja bem-vindo ao primeiro capítulo deste livro. Ele será curto e conterá apenas teoria, mas nos demais compensarei com a prática. Nele buscarei trazer conteúdo sobre o panorama dos dispositivos móveis nos dias de hoje, tipos de dispositivos, suas características e opções para o desenvolvimento de aplicativos.

As ferramentas que serão usadas neste livro serão apresentadas conforme forem necessárias. É importante que você tenha conhecimento do paradigma de Orientação a Objetos e de alguma linguagem de Programação Orientada a Objetos para um perfeito acompanhamento do desenvolvimento proposto para este livro. O desejável é um conhecimento ainda que básico para tecnologias WEB, como HTML, CSS e JavaScript, entretanto, este não é um requisito proibitivo para você ler este livro.

A aplicação proposta para ser desenvolvida neste livro, a partir do terceiro capítulo, refere-se ao atendimento oferecido por uma oficina mecânica. A aplicação subsidiará o cadastro de clientes, peças e serviços a serem contratados para veículos e registro de atendimento a clientes. Nestas funcionalidades, serão trabalhados tipos de telas para a aplicação, controles de entrada de dados, listagem de dados, templates para aparência da aplicação, acesso a banco de dados, consumo de serviços web e uso de câmera.

Não faz parte do escopo do livro ter esta aplicação implementada em sua totalidade, mas sim para a aplicação do Ionic em algumas funcionalidades relacionadas a este problema proposto.

Durante o desenvolvimento da aplicação, diversas técnicas e recursos serão utilizados para subsidiar uma boa arquitetura para o desenvolvimento da aplicação.

1.1 Os dispositivos móveis na atualidade

Antes de começarmos esta seção, permita-me um alerta. A contextualização proposta aqui, é semelhante à que utilizei em demais livros sobre dispositivos móveis aqui da Casa do Código. Optei por isso, por estar trabalhando uma mesma situação. Desta maneira, nesta e na próxima seção, você pode ter um sentimento de *déjà-vu*, mas é só nelas OK? Depois, tudo é com base no Ionic.

Até anos atrás, era comum conhecer pessoas que adquiriam computadores apenas para navegar na internet, ler e-mails e acessar algumas aplicações para leitura de livros, artigos ou documentos diversos. No lado corporativo empresarial, notebooks eram fornecidos aos colaboradores para que desempenhassem algumas atividades, como registro de uma venda para um cliente, recebimento de uma conta, anotação de um pedido e agendamento de compromissos, dentre diversas outras atividades.

Com o surgimento dos dispositivos móveis, a venda de computadores pessoais tem sofrido constantes quedas. Isso começou lá atrás, de maneira modesta e quase despercebida, com handhelds, palmtops e PDAs (*Personal Digital Assistants*), mas que começou a ganhar destaque com a chegada do iPod e depois dos iPhones. A massificação veio por meio do Google, com o desenvolvimento do sistema operacional Android para dispositivos mais **ace$$ívei$**, em relação aos produtos da Apple.

Embora um dispositivo móvel possa permitir acesso para necessidades pessoais e corporativas, é importante ressaltar que a maneira como este mercado se desenvolveu foi bem distinta. Tem-se o lado recreativo, em que é possível ter em seu dispositivo diversos jogos; o cultural, que permite o acesso a filmes e livros; a organização pessoal, com diversos recursos para gestão financeira e de compromissos, acesso a bancos, compras; e a parte empresarial, com aplicativos corporativos, de gestão, operacional das empresas e vendas. Há ainda pessoas que utilizam este

dispositivo apenas para navegar na internet e acessar redes sociais, dispensando, em muitos casos, a necessidade de um computador.

Toda essa popularização e mudança comportamental resultaram no que é visto atualmente como um fenômeno BYOD (*Bring Your Own Device* — Traga seu Próprio Dispositivo). Ou seja, o colaborador está levando para seu ambiente de trabalho um dispositivo potente (o seu) e que pode, em alguns casos, ser substituído por um computador, quer seja desktop ou notebook.

Com isso e o acesso à rede corporativa liberado, todos podem ter no bolso as aplicações de seu local de trabalho. Isso pode gerar mais produtividade também (ou não **:-)**). Porém, no que diz respeito à segurança, traz novas ameaças e vulnerabilidades, mas isso é assunto para a equipe de infraestrutura.

Pense em trabalhos nos quais se exija atividade externa, quer seja em campo ou urbana, como a função de um engenheiro agrônomo ou um vendedor externo. Esses profissionais precisavam se dirigir à empresa, buscar blocos ou impressos específicos para coleta de dados, retornar aos seus escritórios e registrarem os dados coletados. Com um dispositivo móvel, ele não precisa nem ir ao seu escritório, pois seu equipamento já permite o registro de sua coleta ou venda no momento em que ela ocorre. Você também já reparou onde os atendentes em restaurantes registram nossos pedidos? Em um dispositivo móvel.

E se o acesso à internet não for possível no momento da atividade, é possível uma sincronização com os servidores da empresa tão logo uma conexão seja possível. Os dispositivos também podem estar conectados apenas em uma rede interna, não sendo necessária a internet.

As empresas buscam também tirar proveito dessa realidade, quer seja dando maior liberdade para seus colaboradores, ou economizando em despesas como energia, água, condomínio ou aluguel, dentre outras.

Neste livro, trabalharemos aplicações que possam ser executadas, com recursos nativos, nos sistemas operacionais iOS e Android. Neste último, existem diversos dispositivos, nas mais diversas especificações de tamanho e resolução de tela, recursos e processamento.

1.2 O desenvolvimento móvel cross-platform

Não é porque os dispositivos móveis estão em uma grande ascendência que os computadores pessoais se tornaram descartáveis. Existem ainda diversas atividades e aplicativos que precisam ser realizadas e utilizados em um computador que possua um teclado (que não seja na tela), uma tela grande (às vezes mais do que uma) e um desempenho ainda não alcançado pelos dispositivos móveis atuais.

Com isso posto, é importante estar ciente de que as aplicações comerciais trazidas para o ambiente móvel precisam, nas interações com o usuário, ter ou solicitar dados e informações que possam caber "confortavelmente" na tela que será usada. Nada de encher de informações desnecessárias para o processo atual.

O desenvolvimento para dispositivos móveis, até a presente data, vem sendo focado nas plataformas iOS e Android, que são as mais utilizadas. Cada uma dessas plataformas possui suas próprias características e recursos. Com isso, você pensa em escolher uma plataforma para que seu aplicativo funcione e limitar o uso a apenas esta?

Se for uma empresa que especifique a plataforma e o aplicativo deva funcionar apenas entre os colaboradores dela, isso é uma opção sim. Mas e se sua aplicação precisar ser utilizada em plataformas diferentes? Aí esta opção já não poderia ser escolhida.

O que vem ocorrendo no desenvolvimento para dispositivos móveis é que, ao desenvolver uma aplicação para um dispositivo e uma plataforma, esta aplicação possa funcionar em qualquer dispositivo, de qualquer plataforma.

Por exemplo, deseja-se que uma aplicação desenvolvida para o iPhone XS Max possa funcionar da mesma maneira em um smartphone com Android. O que fazer para resolver este problema? Montar várias equipes de desenvolvimento, cada uma usando um ambiente de desenvolvimento diferente e focada em uma plataforma? Desenvolver diversas versões do mesmo aplicativo? Isso não é produtivo, e nem barato, concorda?

O objetivo é partir para uma ferramenta que possibilite o desenvolvimento *cross-platform*. Existem várias. Algumas geram o aplicativo para ser executado em um ambiente específico, como se fosse uma máquina virtual (conhecido como ambiente híbrido). Outras ferramentas geram código para ser executado em um ambiente web, como se fosse um site em um navegador (que é o caso do Ionic) e, temos ainda, ferramentas que geram aplicativos nativos para as plataformas escolhidas, como o Xamarin. O Ionic, por meio de frameworks específicos, como `Cordova` e `Capacitor`, possibilitam a `transformação` de uma aplicação Ionic em uma aplicação híbrida. Veremos isso a partir do próximo capítulo.

O site do Ionic oferece um livro gratuito chamado `Hybrid Vs. Native`, que você pode obter em https://ionicframework.com/resources/. Quando tiver tempo, talvez seja interessante lê-lo. É bem pequeno.

Um ponto importante é verificar na App Store (iOS) e no Google Play os padrões e as regras que devem ser respeitados para a publicação oficial de sua aplicação. Por exemplo, os ícones das aplicações são diferentes em cada plataforma, assim como a barra de navegação e o padrão de cores. São alguns dos pontos que você deve verificar e garantir que sua aplicação os respeite para que a publicação seja aceita. E para você distribuir aplicações pela App

Store e Play Store, será necessário ter uma conta de desenvolvedor, que é paga.

1.3 O Ionic Framework

De acordo com o próprio portal do Ionic, ele é definido como um framework, de código aberto, de ferramentas para serem utilizadas para o desenvolvimento da interface com o usuário, que permite um bom desenvolvimento de aplicações baseadas em tecnologias para web, como JavaScript, HTML e CSS. Outra definição, no próprio portal, traz o Ionic como uma biblioteca de componentes de interface com o usuário (UI).

Embora o Ionic seja *open source* e permita o desenvolvimento de aplicações com ferramentas gratuitas, ele possui pacotes com recursos adicionais e que são pagos - e que talvez você julgue interessante, quando puder, verificar em https://ionicframework.com/pro/pricing/.

Como dito no primeiro parágrafo desta seção, o Ionic tem como foco o desenvolvimento de aplicações baseadas em tecnologia para web, que executam em um navegador, ou ainda em dispositivos de plataforma específica. Desta maneira, o desenvolvimento cross-platform por meio do Ionic faz uso do modelo híbrido de aplicações móveis. Ou seja, ele permite que aplicações desenvolvidas em Ionic sejam executadas em dispositivos móveis, mas não fazendo uso direto de recursos nativos, embora, como veremos, isso seja possível, por meio de componentes de terceiros. As aplicações Ionic são conhecidas como `Webview`, ou ainda, `Frontend Application`.

Como uma plataforma para desenvolvimento de aplicativos cross-platform, o Ionic tem como foco dispositivos móveis com o iOS e Android e, quando executarmos nossa primeira aplicação, a

veremos em seus respectivos pseudoemuladores, gerados pelas ferramentas do Ionic.

A linguagem de programação que adotaremos e a padrão para o desenvolvimento em Ionic é a TypeScript, que possibilita um desenvolvimento estático, com compilação em tempo de implementação, garantindo que o código JavaScript que será gerado esteja livre de erros sintáticos. Também utilizaremos o HTML e CSS em nossas camadas de apresentação e você verá que estes recursos abrem um enorme leque de customização da interface com o usuário. Estes conhecimentos não são excludentes para quem deseja ler este livro, pois fornecerei explicações que sejam suficientes para o acompanhamento das atividades, buscando sempre deixar referências que possam complementar seu conhecimento.

O TypeScript, de acordo com seu próprio portal, é um superconjunto tipado que é compilado para JavaScript puro. Todo código TypeScript que implementarmos, após uma compilação bem-sucedida, será gerado para código JavaScript, que é o código que será utilizado pela aplicação no dispositivo. Embora a linguagem foco seja TypeScript, teremos implementações em JavaScript puro também, o que é perfeitamente válido para código TypeScript.

O TypeScript foi desenvolvido como uma iniciativa da Microsoft, por Anders Hejlsberg, criador do Turbo Pascal, Delphi e C#. Vem sendo uma linguagem/ferramenta adotada de forma cada vez maior pela comunidade de desenvolvedores JavaScript. Ela permite inclusive o desenvolvimento orientado a objetos, fazendo uso de classes, interfaces, herança e métodos, dentre outras características deste paradigma. Conforme formos utilizando esses recursos durante nossas aplicações, procurarei contextualizá-los.

De acordo com relatos identificados na web, o Ionic sofreu mudanças drásticas na passagem da versão 1 para a 2 e, consequentemente, a 3. Essa atitude causou uma série de desconfortos na comunidade desenvolvedora, pois gerou muita mão

de obra para atualização de suas aplicações, havendo casos até de opções por mudança de plataforma. No momento em que decidi escrever este livro, o Ionic estava em sua versão 3, mas já com a 4 em situação de Beta 4. No momento final do livro, a versão 4 já está disponível e estável e é nessa nova versão, a 4, em que teremos nossas implementações realizadas.

Com a chegada da versão 4, houve também alterações, principalmente na estrutura dos projetos. Estas mudanças, a princípio, são causadas pela forte dependência do Ionic com o Angular, o que a partir desta nova versão diminuirá, e o compromisso deles é o desenvolvimento não só utilizando o Angular, mas também o Vue e o React. Nós utilizaremos o Angular neste livro. No próximo capítulo comentarei ele de maneira mais descritiva.

Conclusão

Conforme prometido no início deste capítulo, ele foi breve. Eu poderia trazer muito mais teoria aqui, mas penso que você quer começar logo com a prática. Entretanto, precisava lhe apresentar o texto aqui trabalhado.

É importante ler um pouco sobre dispositivos móveis, cross-platform, TypeScript, JavaScript, HTML e CSS. É claro que o que eu trouxe para você foi apenas uma introdução, mas acredite, com ela você já está pronto para começar o desenvolvimento móvel e tem motivos para escolher o Ionic. Isso porque você quer e precisa desenvolver para, no mínimo, duas plataformas e posso assumir que não quer, ou não pode, ter uma equipe para cada plataforma.

Você precisa de produtividade e quer que sua aplicação seja desenvolvida em uma única plataforma, mas que seja executada nas duas mais usadas atualmente. Então, a solução que proponho neste livro é o Ionic.

No próximo capítulo, instalaremos a ferramenta e testaremos seu funcionamento. Até lá.

Capítulo 2
Ionic Instalação e testes

O processo de preparação para o ambiente de trabalho para desenvolvimento e distribuição de aplicações Ionic em dispositivos móveis exige a instalação de algumas ferramentas que são pré-requisitos. Neste segundo capítulo veremos estas ferramentas e algumas possibilidades para execução e testes de aplicações. Também instalaremos o ambiente para a codificação de nossas práticas, o IDE, que será o Visual Studio Code. Com as ferramentas e dependências instaladas, trabalharemos a criação e execução de uma aplicação simples, para validar a corretude desse processo. Trabalharemos a distribuição da aplicação em dispositivos físicos e emuladores, por meio de frameworks que realizam esta ligação de código front-end com os SDKs nativos de cada plataforma. Veremos o uso do Cordova e Capacitor. Temos aqui um capítulo relativamente simples, mas que envolve um certo esforço e atenção em todo o processo.

2.1 Ferramentas, downloads e instalação

Estou usando dois equipamentos para desenvolvimento. Um i7, com 8GB de RAM e Windows 10 com todos os updates realizados e um MacBook Pro, i5, com 4GB de RAM, executando o macOS Mojave 10.14.4, devidamente atualizado. Todas as ferramentas utilizadas nestes equipamentos podem também ser instaladas em plataforma Linux, seguindo sempre as orientações específicas para ela, que são sempre fornecidas nos portais destas ferramentas.

Como dito no início do capítulo, para instalarmos o Ionic e o termos disponível para nosso desenvolvimento, precisamos antes realizar o download de algumas dependências, e a primeira delas será o `Git` ,

que é um sistema de controle de versões distribuído. O Git é muito usado entre desenvolvedores, podendo versionar qualquer tipo de documento. Em conjunto com o Git, é muito comum a associação com o GitHub, portal que hospeda um imenso número de projetos, componentes, aplicações e documentos.

É possível obter o Git para download em https://git-scm.com/downloads/ e, no momento da escrita deste livro, a versão disponível é a 2.21.0. Realize o download e instalação deste nosso primeiro pré-requisito. Caso não queira se preocupar, basta seguir as sugestões do padrão de instalação. Se quiser se aprofundar em cada opção, recomendo uma leitura no portal da ferramenta.

Se você estiver em uma rede com proxy, pode ser que seja necessário realizar algumas configurações, que são apontadas no link https://ionicframework.com/docs/cli/configuring.html/, mas trago de maneira resumida na sequência.

```
// Variáveis de ambiente para o GIT
git config --global http.proxy http://<proxy>:<porta>
git config --global https.proxy http://<proxy>:<porta>

// Retirando o proxy do GIT
git config --global --unset http.proxy
git config --global --unset https.proxy

// Configurações de variáveis de ambiente para o Ionic
HTTP_PROXY=http://<proxy>:<porta>
HTTPS_PROXY=http://<proxy>:<porta>
IONIC_HTTP_PROXY= http://<proxy>:<porta>
```

Também teremos necessidade do Java (JDK) instalado em nossa máquina, pois desenvolveremos para Android e o Java é um pré-requisito. Embora no momento de escrita do livro exista a versão 12.0.1 do JDK, o Cordova na versão que utilizaremos, precisa da 1.8.X, que você pode baixar em https://www.oracle.com/technetwork/pt/java/javase/downloads/jdk8-downloads-2133151.html. Será preciso que você coloque o caminho

no PATH de sua máquina, incluindo a pasta `bin` e você pode ver orientações para isso aqui: https://www.java.com/pt_BR/download/help/path.xml.

Para trabalharmos com a plataforma Android, quer seja por meio de dispositivos ou emuladores, eu particularmente recomendo a instalação do Android Studio, pois ele faz todas as configurações necessárias para nosso desenvolvimento. É possível instalar apenas os recursos necessários caso você queira, mas por dificuldades que tive em relação a isso, buscando uma solução mais simplista e efetiva, adotei a instalação desta ferramenta, que pode ser obtida em https://developer.android.com/studio que, no momento da escrita deste livro, está na versão 3.4.1.

Ao executar o instalador, se você estiver em uma rede com proxy, o instalador solicitará que a configuração seja realizada, pois ela depende da conexão com a internet. Optei pela detecção automática e isso foi suficiente.

Com o término da instalação, execute o `Android Studio` e na janela que se abre, selecione o botão `Configure` e nele a opção `SDK Manager`, tal qual mostra a figura a seguir.

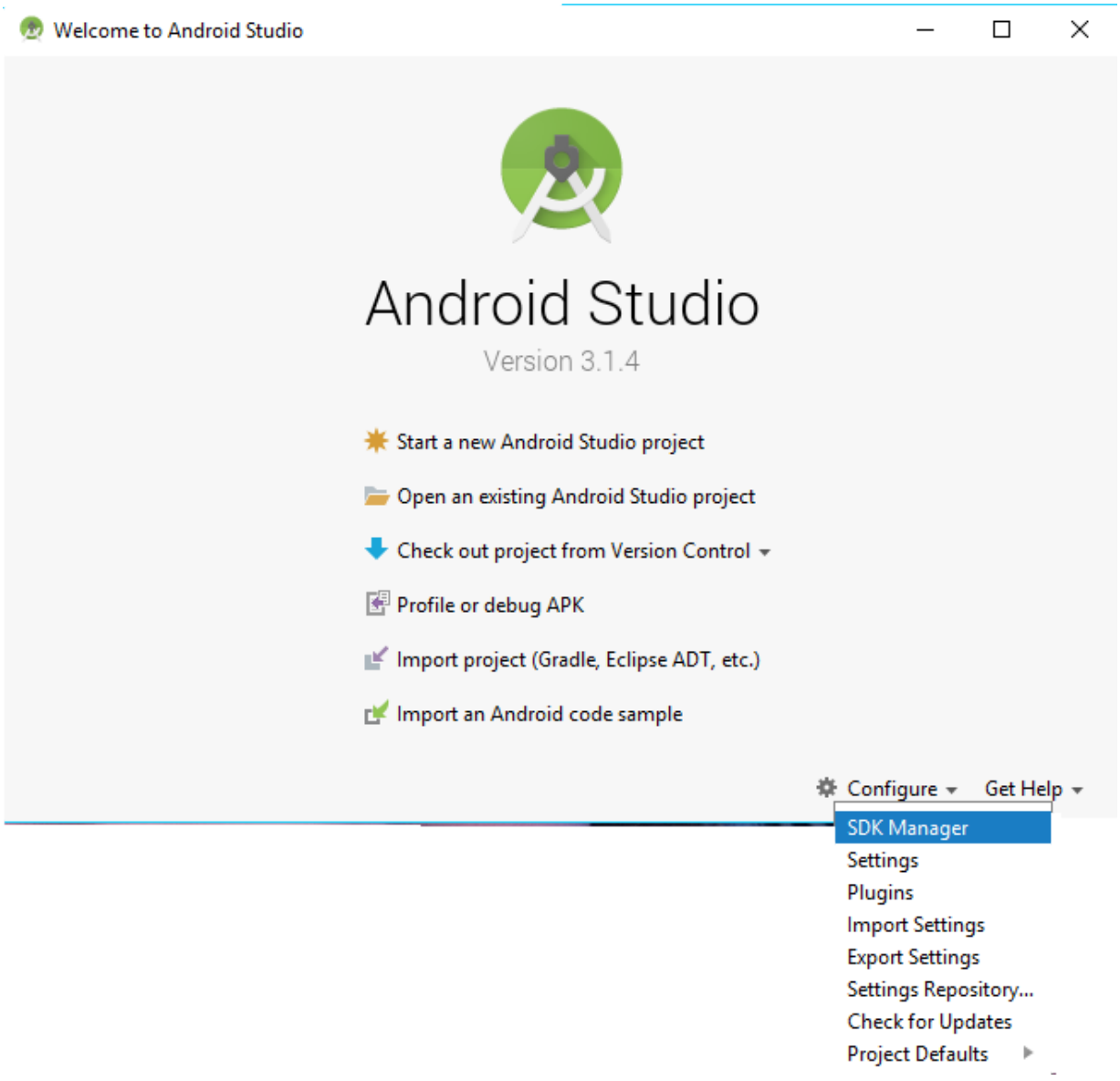


Figura 2.1: Acessando o gerenciador do Android SDK

Na janela que se abre, serão exibidas as plataformas disponíveis para instalação. Eu recomendo que você instale as três últimas versões ao menos, em meu caso foram as APIs 28, 27 e 26. Esta decisão se deu devido a um erro no processo de preparação da aplicação para o Android, pelo Ionic CLI, que solicitou a API 26 e a 27, por um problema no uso do Capacitor, que veremos mais adiante. E antecipando estes problemas, que acredito ser um bug, que possa estar corrigido no momento em que você está lendo este

livro. Após a seleção, confirme a instalação clicando no botão `Apply`. Este processo pode demorar um pouco, pois é dependente de conexão com a internet. Observe e guarde o endereço, que aparece no topo desta janela, que é onde o SDK foi instalado. Precisaremos dele. Nesta janela você também tem a opção de configuração de proxy.

Precisamos agora baixar e instalar em nossa máquina o(s) emulador(res) onde executaremos nossa aplicação Android. Como baixamos o Android Studio, vamos fazer uso dele para este processo. Da mesma maneira que acessamos o `SDK Manager` anteriormente, vamos acessar o `AVD Manager` para esta atividade. Ao acessar esta ferramenta você terá acesso a uma janela como a apresentada na figura a seguir.

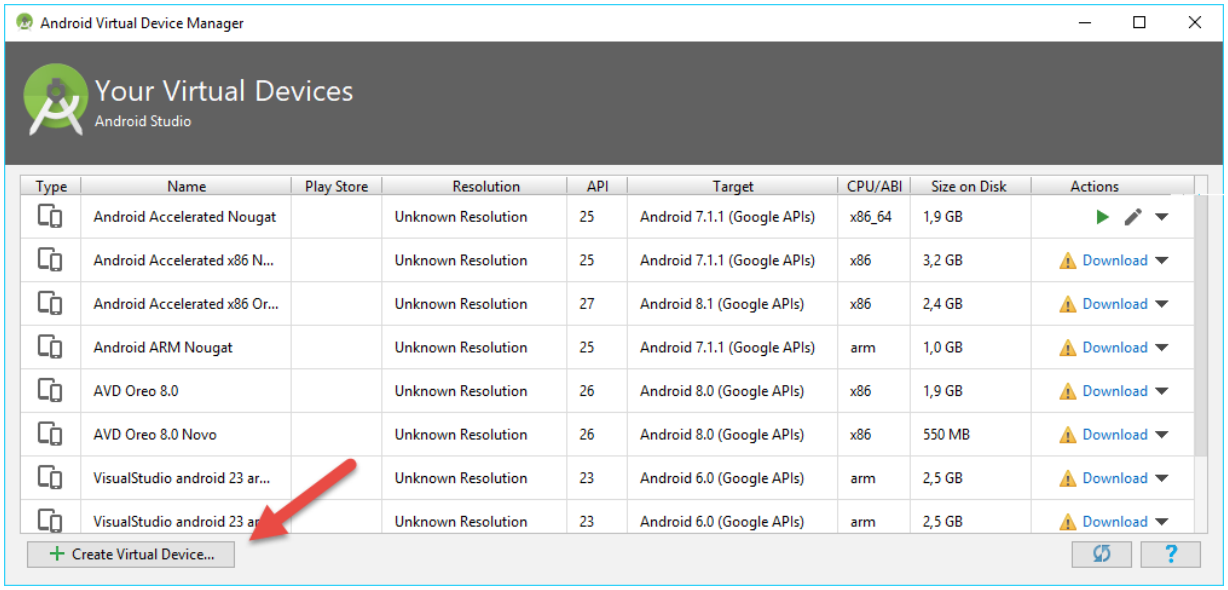


Figura 2.2: Acessando o gerenciador de dispositivos virtuais do Android

Nesta janela, existe uma relação de dispositivos previamente configurados e que nos permitem o download para os termos em nossa máquina. Pode ser que nenhum dispositivo seja apresentado em seu AVD Manager, o que nos remete a criá-lo. Para isso, clique no botão `Create Virtual Device`. Na janela que se abre, precisamos escolher o modelo do dispositivo que utilizaremos. Eu selecionei o

`Nexus 6P`, dentro da categoria `Phone` e então cliquei no botão `Next`. A nova janela traz três guias. Na primeira, precisamos instalar a API para a imagem do dispositivo que estamos criando. Eu selecionei a 28 neste exemplo e realizei o download dela. Ao término do download, precisamos ir para a segunda guia, responsável pela imagem em si do dispositivo. Optei pela x86_64 da API 28 e fiz o download dela. Após a conclusão, retornei para a primeira guia e cliquei no botão `Next`, o que levou para a última janela, para definição do nome do emulador, informe o que desejar. Confirme a operação e você retornará para o AVD Manager, podendo executar o emulador.

Com a configuração dos emuladores concluída, precisamos criar uma variável de ambiente, chamada `ANDROID_SDK_ROOT`, apontando para o SDK do Android, que em minha máquina é `C:\Users\Everton\AppData\Local\Android\Sdk`. Este é o endereço que comentei há pouco para você guardar. Uma vez mais, é possível verificar como criar uma variável de ambiente em https://www.java.com/pt_BR/download/help/path.xml/.

Começaremos agora a instalação de ferramentas que usaremos diretamente em nosso desenvolvimento e começaremos com o `NodeJS`. De maneira simplista e rápida, podemos dizer que o `Node.js` é um aplicativo interpretador de código JavaScript, executando no lado servidor da aplicação e não (apenas) no lado cliente, como é comumente utilizado. Segundo o próprio portal do Node.js (https://nodejs.org), a ferramenta tem também o objetivo de auxiliar programadores na criação de aplicações escalonáveis, com enorme capacidade de processamento. Durante a leitura deste livro, faremos uso do `Node` nos momentos em que instalaremos plugins e componentes que nos auxiliem no desenvolvimento, além do próprio Ionic. Também teremos o Node em execução quando formos testar as aplicações que desenvolveremos em Ionic. A versão disponível no momento da escrita deste livro é a 12.3.1.

Você pode verificar se já tem o `Node` instalado em seu ambiente, bastando, no console (`Command`, `Windows PowerShell` como

Administrador sempre, ou `Terminal`) digitar `node -v` e `npm -v`. Caso as ferramentas já estejam disponíveis em sua máquina, verifique as informações retornadas e veja se seu ambiente está atualizado. Caso precise atualizar, recomendo baixar novamente o instalador e executá-lo, é mais simples. Entretanto, existe na documentação processos apenas para a atualização, mas deixarei isso para você.

Em relação ao NPM (Node Package Manager - Gerenciador de Pacotes do Node), ferramenta que utilizaremos muito, o portal NodeBR (http://nodebr.com/) traz a informação de que ele pode ser visto de duas maneiras: a primeira, e segundo o portal, a mais importante é que o NPM é um repositório online para publicação de projetos de código aberto para o Node.js; em sua segunda visão, mais técnica, é um aplicativo de linha de comando que interage com este repositório online, auxiliando na instalação de pacotes, gerenciamento de versão e gerenciamento de dependências em nossas aplicações. Quando instalamos o Node.js, o NPM é também instalado por padrão.

Retornando ao Ionic, podemos afirmar que as aplicações desenvolvidas por ele são criadas e assistidas por um utilitário de linha de comando, conhecido como `Ionic CLI`, ou Ionic Command Line Interface. O Ionic CLI é a ferramenta que o framework oferece para execução de diversas atividades durante o processo de desenvolvimento, como a criação do projeto da aplicação, criação automatizada de artefatos para seus projetos e execução das aplicações, dentre outras funcionalidades que veremos no desenvolver do livro.

Para instalarmos o Ionic e termos a nossa disposição o Ionic CLI, precisaremos fazê-lo por meio do NPM, que já instalamos em passos anteriores, junto com o Node.js. Para iniciarmos, é preciso acessar a console do Windows (Command) ou o `Windows PowerShell`, com direitos de administrador. Já no Mac é preciso acessar como `sudo`. Veja o conjunto de instruções a seguir, cada um precedido de comentários. Algumas instruções são específicas para a situação de estarmos em uma rede com proxy.

```
// Configuração de proxy para o NPM, se estiver em uma rede onde exista um
npm config set proxy <proxyServer>:<port>
npm config set https-proxy <proxyServer>:<port>

// Instalação da versão mais atualizada do Ionic de maneira global no
sistema (-g)
npm install -g ionic@latest

// Instalação de uma versão específica para o Ionic, no caso, a 5.0.1, que
será a que utilizaremos em todo o livro
npm install -g ionic@5.0.1

// Desabilitação do proxy
npm config delete proxy
npm config delete https-proxy
```

É interessante ter o conhecimento de que, ao se configurar o proxy com as instruções anteriores, um arquivo chamado `.npmrc`, que fica na pasta `\users\<usuário>` é criado/atualizado com estas configurações. Você pode optar por utilizar este arquivo em vez de executar o NPM. Fica a seu critério.

A versão instalada em minha máquina para a implementação dos exemplos deste livro foi a mais atualizada no momento da escrita, que foi a 5.0.1. Você pode verificar a versão do Ionic em sua máquina executando, no terminal, a instrução `ionic --v`. Recomendo que mantenha esta versão durante todo o livro. Caso opte por versões diferentes, mais recentes ou anteriores, podem surgir problemas relativos a especificidades de cada uma.

Com o Ionic e suas dependências devidamente instaladas, precisamos agora do ambiente que utilizaremos para o desenvolvimento de nossas aplicações e adotaremos neste livro o `Visual Studio Code`, que tem sido adotado amplamente pela comunidade de desenvolvedores, de diversas linguagens. Ele é gratuito e realmente muito bom. Você pode realizar seu download pelo link https://code.visualstudio.com/. No momento da escrita deste livro a versão foi a 1.34.0.

Após a instalação e execução do aplicativo, é possível verificarmos atualizações do ambiente pelo menu `Help -> Check for updates`. Em caso de existência de atualizações necessárias, aparecerá o item de menu `Install updates`. Basta seguir as orientações para que seu ambiente seja atualizado.

Como dito no capítulo anterior, utilizaremos o TypeScript como linguagem, pois é a linguagem adotada pelo Ionic. Também foi comentado que o código TypeScript é convertido (automaticamente) para JavaScript (processo de `Transpiling`), que é a linguagem que realmente terá o código executado e, certamente utilizaremos os recursos dela, em conjunto com o TypeScript. Desta maneira, é preciso que nosso ambiente (IDE) seja configurado para estas linguagens, o que nos trará benefícios, como *autocomplete* e *intellisense*. Sendo assim, na primeira execução do ambiente, aparecerá na janela principal a sugestão de instalação de recursos para estas linguagens. Clique nos links oferecidos para que isso seja realizado. Veja a figura a seguir. A guia apresentada pode ser acessada por meio do menu `Help -> Welcome`. Já já começaremos a utilizar este ambiente.

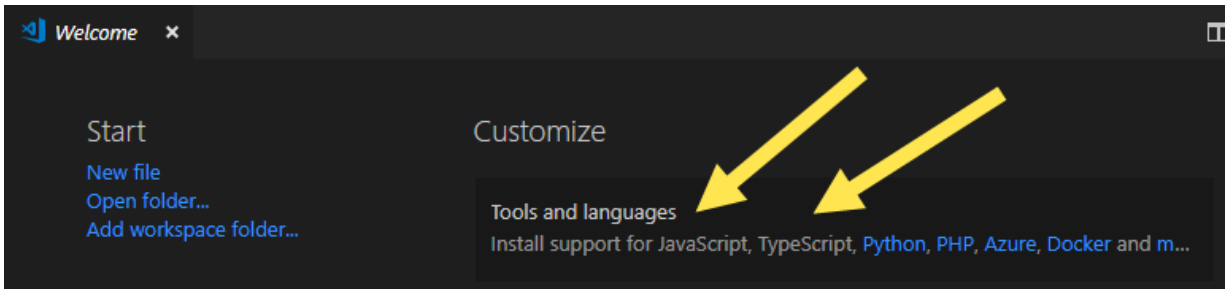


Figura 2.3: Intalando suporte ao TypeScript e JavaScript

A título de curiosidade, você pode pensar em obter e avaliar o Ionic Studio em https://ionicframework.com/studio/. Ele não é gratuito, mas a licença individual não é cara. Experimente, e se gostar, pense em adquirir.

2.2 Testando a instalação do ambiente com um projeto básico

O Ionic fornece diversos modelos (templates) de projetos que podemos utilizar durante o processo de criação e você pode obter uma lista completa deles executando a instrução a seguir no terminal.

```
ionic start --list
```

Vamos trabalhar a criação de um projeto básico (`blank`), mas recomendo que você teste a criação de projetos para todos os templates disponíveis, pois isso poderá trazer a você uma visão de alguns recursos que poderão ser utilizados em suas aplicações. Durante o desenvolvimento de exemplos do livro criaremos as funcionalidades oferecidas por alguns deles.

Para a criação do projeto, recomendo que criemos uma pasta/diretório na raiz de seu disco e então nela execute, estando no terminal, a instrução a seguir. Este processo pode demorar um pouco, pois depende de sua conexão com a internet, uma vez que o `Ionic CLI` realizará o download do template desejado. Aqui houve uma situação de problemas com a versão atualizada do Node.js executando em um ambiente Windows 7. A pasta era criada, mas não começava o processo de download. Precisei baixar uma versão anterior, no caso a 8.11.4 para que o projeto fosse criado. Veja na instrução que `capitulo02` é o nome do projeto, `blank` é o template que será utilizado, `type` define o `angular` como engine de templates e, no início, `start` define que o projeto deve ser criado.

```
ionic start capitulo02 blank --type=angular
```

Logo no começo da execução do processo disparado pela instrução anterior, uma mensagem de criação da pasta para o projeto será exibida. Caso você já tenha a pasta, será perguntado se deseja sobrescrever a que existe. Caso responda negativamente, a criação do projeto é interrompida. Durante a criação do projeto algumas

mensagens, orientações e sugestões serão exibidas. Ao final, uma pergunta sobre a instalação do `Ionic Pro SDK` é realizada. Responderemos com `N` neste momento. Observe que durante a instalação é informado que um repositório local do `Git` é criado. Poderíamos ter utilizado, na instrução de criação do projeto o flag `--no-git` para que isso não ocorresse.

Você notou, na instrução para a criação do projeto, a definição do tipo como Angular (`--type=angular`)? Pois bem, como dito no capítulo anterior, o Ionic tem uma forte dependência com o Angular, mas a partir desta nova versão, as aplicações poderão fazer uso do `Vue` e `React` . Mas o que são estes artefatos?

Bem, vamos falar do Angular. Segundo seu próprio site (https://angular.io/), ele é uma plataforma que auxilia no desenvolvimento de aplicações web, tendo um foco dedicado às `Simple Page Applications` , que são portais que possuem uma única URL e toda a informação dinâmica dela é configurada por meio de CSS e JavaScript. É uma ferramenta muito boa, poderosa, com muitos recursos e que vale a pena um estudo, principalmente, em nosso caso, pelo fato do Ionic fazer uso destes recursos. Nosso foco, neste livro, é o Ionic, mas sempre que formos utilizar recursos vindos do Angular, manifestarei esta situação.

Em nosso projeto temos o arquivo `package.json` , que traz todas as dependências necessárias para nosso projeto. As que são definidas por padrão e as que podemos instalar no futuro. Ao abrir este arquivo e buscar por `"@ionic-native/core"` , deparei-me com a versão `5.0.0` , que não era a mais recente na época. Isso me levou a executar a instrução a seguir, para ter em meu projeto a versão mais recente deste pacote e essa versão, a `5.7.0` que utilizaremos no livro.

```
npm install --save @ionic-native/core@5.7.0
```

O `Ionic Native` é uma biblioteca de plugins Cordova que possibilita de maneira simplificada a adição de funcionalidades nativas à

plataforma em execução de aplicações móveis e, por isso, é bom termos a versão atualizada em nossos projetos.

Voltemos ao projeto. Caso você queira fazer um backup de sua aplicação, você pode retirar a pasta `node_modules`, criada dentro de `capitulo02`, pois ela é imensa e pode ser gerada automaticamente por meio do NPM. Quer testar? Crie uma pasta chamada `teste`, no mesmo nível de `capitulo02` e copie para ela todo o conteúdo de `capitulo02`, com exceção da `node_modules`. Acesse a nova pasta pelo terminal e execute a seguinte instrução e, ao término, verifique que a pasta `node_modules` foi criada. Se quiser, pode apagar a pasta `node_modules` do `capitulo02` e executar a mesma instrução para instalar as dependências.

```
npm install
```

Mas o que é o conteúdo desta passa `node_modules`? Ainda, o que são as dependências de que estamos falando de maneira constante neste início de livro? Você certamente notou que para utilizarmos o Ionic, tivemos que instalar alguns recursos e que, ao criarmos um projeto, muitos recursos específicos para um projeto Ionic, foram baixados e disponibilizados na pasta `node_modules`. Estes recursos são vistos como dependências, sem as quais sua aplicação não pode ser desenvolvida e executada. Reforçando, de maneira mais direta, aplicações Ionic dependem muito de outras ferramentas, que podem ser aplicações, componentes e plugins. Veremos e trabalharemos estas dependências durante toda a leitura do livro.

Agora que já temos nosso projeto criado, vamos testá-lo? Acesse a pasta do projeto, que seguindo nosso exemplo, deve se chamar `capitulo02`. Você pode acessá-la digitando `cd .\capitulo02\` no Windows PowerShell. Estando na pasta, digite a instrução a seguir.

```
ionic serve
```

Você poderá notar a exibição de algumas informações ao executar a instrução anterior. As informações iniciais se referem a como acessar a aplicação. Note que a primeira instrução executada é uma

chamada ao `Angular CLI`, o `NG`. Na sequência, uma instância de seu navegador padrão será aberta, apresentando a aplicação, tal como é mostrado na figura a seguir. Para registro, eu tenho utilizado o Chrome como navegador padrão, por ser, de acordo com a comunidade, o navegador que está sempre atualizado com as tecnologias web.

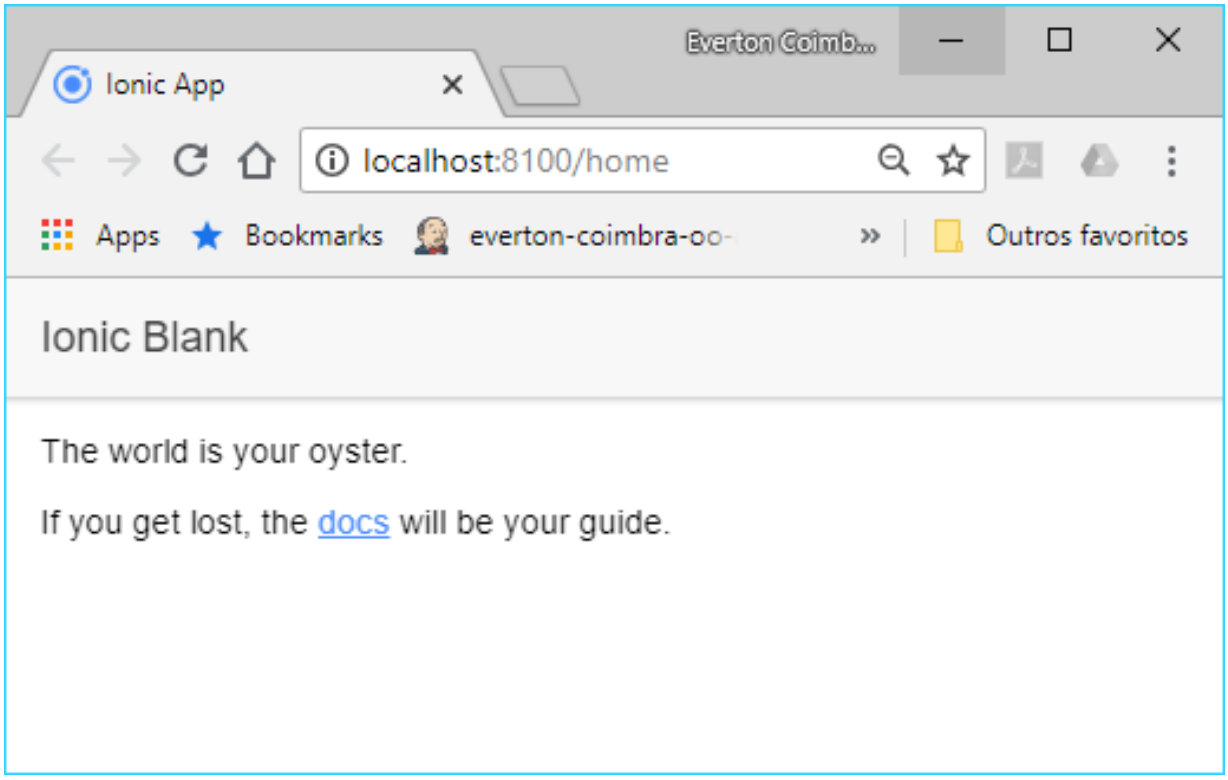


Figura 2.4: Execução básica da aplicação

Olhando para a figura anterior, ou para seu navegador, você pode se perguntar, mas é assim que a aplicação executará em meu dispositivo móvel? Vamos por partes. Lembre-se de que as aplicações Ionic são Webview, ou seja, são executadas em um ambiente de um navegador web. Então, até o momento, está tudo OK com a execução da aplicação.

Retorne ao terminal, veja que várias instruções apareceram nele. São todas informações de LOG, geradas pela preparação e execução da aplicação. Mas vamos verificar agora a execução da

aplicação nos pseudoemuladores, que comentei no capítulo anterior. No terminal, pressione `CTRL-C` para interromper a execução. Pode ser que precise pressionar estas teclas algumas vezes. No prompt, digite a instrução a seguir.

```
ionic serve --lab
```

O flag `--lab` indicará ao Ionic que a execução da aplicação, embora em um navegador, deve ser apresentada em um ambiente de dispositivo móvel. Ainda, ao executar, é possível que o Ionic retorne a você a informação de que não tem, nas dependências do projeto, o pacote `@ionic/lab` e se ofereça para instalá-lo. Vamos confirmar com `Y` caso isso ocorra. Após todo o processo de execução e instalação do pacote, o navegador é novamente aberto, agora exibindo a aplicação nos pseudoemuladores.

Permita-me apresentar um erro que pode acontecer com você quando for executar a instrução que habilita os pseudoemuladores. Em um determinado momento, quando executava a instrução anterior, nada aparecia no navegador. Parti para a inspeção de código do Chrome, pressionando `F12` , ou clicando com o botão direito do mouse em qualquer lugar da página e escolhendo a opção `Inspecionar` . A janela do navegador se divide, e no lado direito é exibida uma poderosa ferramenta para depuração. Cliquei na guia `Console` e vi lá um erro com a descrição `Failed to load resource the server responded with a status of 404 (not found)` . Interrompi a execução do Ionic CLI e, na pasta do projeto, digitei `npm install` , para forçar o download das dependências dos módulos node. Lembra que falei dessa instrução um pouco antes no texto? Após a instalação, executei novamente a instrução `ionic serve --lab` e tudo voltou a funcionar. Em outro caso, onde esta estratégia não funcionou, eu forcei a carga da página no navegador, com `CTRL+F5` e tudo funcionou. Veja que esta segunda é mais simples e deve ser a primeira a ser tentada.

Após este comentário, vamos ao navegador aberto. Veja a figura a seguir. Talvez apareça para você o pseudoemulador do Windows,

mas eu preferi tirá-lo, por meio do link `Platforms`. Esta opção se deu pela dificuldade de encontrar dispositivos Windows para testes físicos. Recomendo que deixe sempre apenas um pseudoemulador ativo, é mais rápido. Habilite mais de um apenas quando precisar ver a interface que o usuário visualizará.

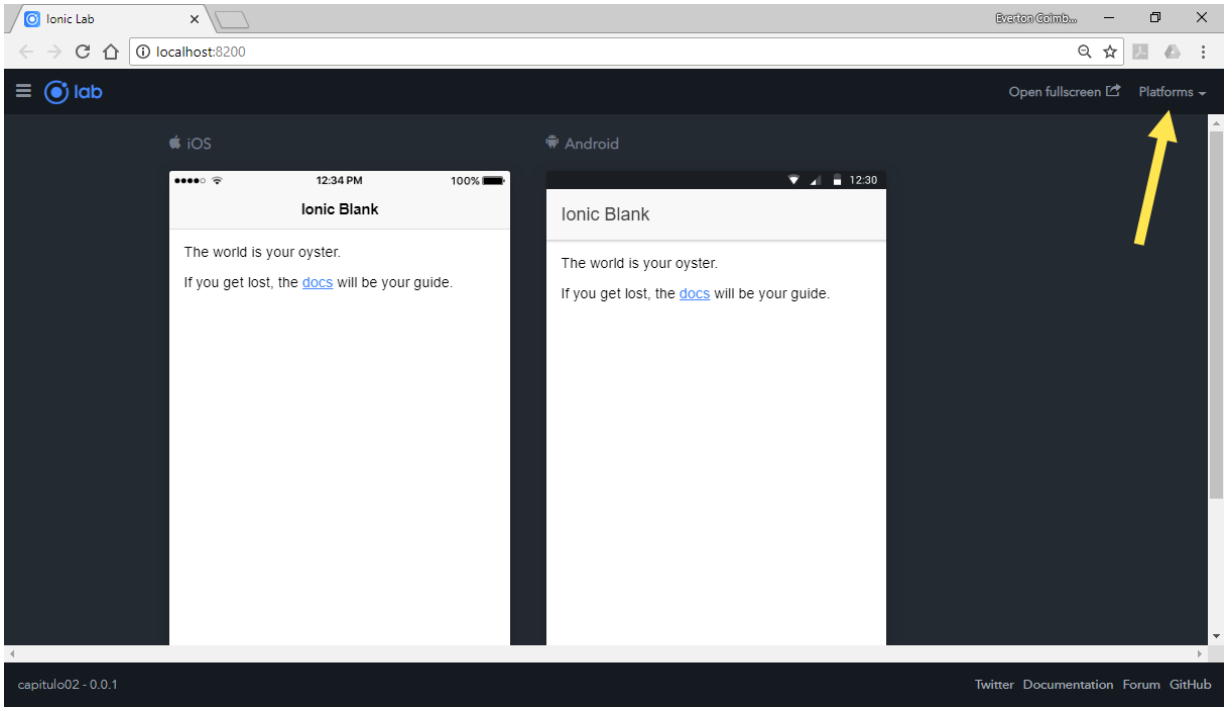


Figura 2.5: Execução da aplicação nos pseudoemuladores

2.3 Execução de forma nativa no iOS com o Cordova

Lembra-se de que comentamos sobre aplicações Ionic poderem ser híbridas? Isso é possível por meio do uso de frameworks que permitem a comunicação de nossa aplicação com o SDK Nativo. Estas aplicações são chamadas de `Native Bridge`. Vamos então testar a execução em emuladores e dispositivos reais? Vamos começar com o iOS, que tem a necessidade de termos nosso projeto sendo desenvolvido em um equipamento Mac. Mesmo que

você não vá testar sua aplicação em um Mac, peço que leia toda esta seção, por ter procedimentos que serão importantes também para Android.

O Ionic se integra com duas plataformas que possibilitam o acesso nativo, são elas o `Capacitor` (https://capacitor.ionicframework.com/) e o `Cordova` (https://cordova.apache.org/), que é o mais conhecido, mas o Capacitor tem prometido vanguarda.

Vamos preparar, inicialmente, nosso projeto para o Cordova, que ainda não temos instalado em nossa máquina. Anteriormente, quando instalamos o Ionic, poderíamos ter instalado diretamente o Cordova, adicionando `cordova` à instrução. Podemos realizar a instalação dele de maneira independente, por meio da instrução a seguir. Lembre-se de usar o `sudo` no Mac. Note que também temos uma instrução que habilitará a integração de nosso projeto com o Cordova. No momento da escrita deste livro, a versão do Cordova é a 9.0.0.

```
// Instala o Cordova no ambiente da máquina.
npm install -g cordova

// Instala a integração de seu projeto com o Cordova. O --add é para a
criação do arquivo `config.xml`, caso não exista no projeto.
ionic integrations enable cordova --add
```

Precisamos agora realizar algumas configurações para execução do projeto. Vamos então abri-lo no Visual Studio Code. É importante saber que o ambiente de trabalho de um projeto neste IDE é a pasta onde o projeto está instalado. Desta maneira, precisamos abrir esta pasta no VSC. Se você está com a guia `Welcome` aberta, clique no atalho `Open Folder` , ou, se preferir, acesse o menu `File -> Open Folder` e localize a pasta para abri-la. Veja a figura a seguir, que apresenta o ambiente com a pasta aberta.

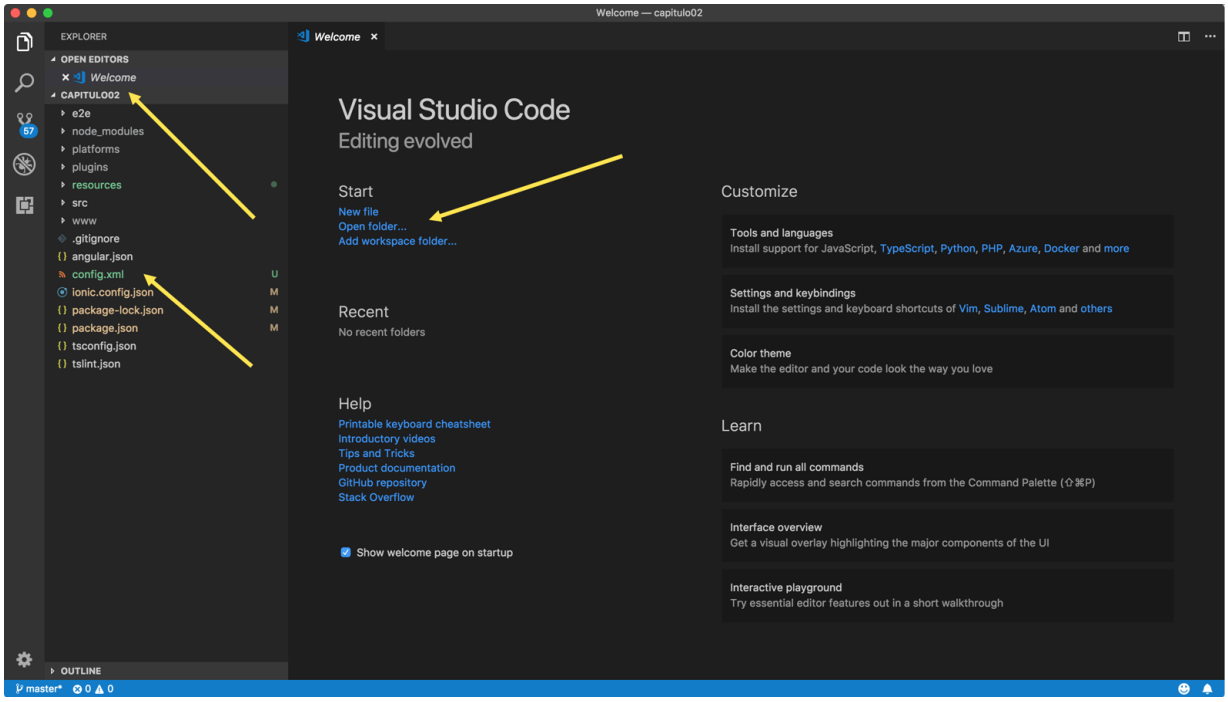


Figura 2.6: Visual Studio Code com a pasta do projeto aberta

Com a pasta aberta, selecione o arquivo `config.xml` na raiz do projeto para que o VSC o abra em seu editor de código. Este arquivo foi criado pela instrução de integração, que executamos anteriormente. Precisamos agora alterar o atributo `id` do `<widget>` que aparece logo no início do código. O valor que devemos ter neste atributo deve respeitar uma notação chamada `Reverse domain name notation`, que traz o domínio da aplicação iniciando pelo fim. Por exemplo, teremos nosso domínio chamado `casadocodigoionic.com.br`. Assim, nosso `id` será `br.com.casadocodigoionic`. Vamos aproveitar também para alterar a propriedade `<name>`. Eu utilizei `Capitulo02`, apenas para ilustrar.

Agora, com o `Cordova` disponível em nosso ambiente (global e local), precisamos preparar nosso projeto para ser executado na plataforma iOS. Desta maneira, no terminal e na pasta do projeto, digite as instruções a seguir. Este processo pode demorar um pouco. Durante a preparação iniciada, o Ionic CLI poderá informar que a plataforma iOS não está instalada, podemos responder com `Y` neste momento, para realizarmos a instalação necessária.

Podemos usar o Terminal diretamente do VSC, bastando acessar o menu `Terminal -> New Terminal`.

Aqui, uma dica. Ao acessar o Terminal, verifique o caminho atual, se é o mesmo, pois, caso não seja, o `npm run build` gerará erros. É importante que, em caso de algum erro nos procedimentos a partir deste ponto ocorra, que você continue lendo, pois após as instruções comento sobre possibilidades que possam ocorrer e como tratá-las.

```
// Esta instrução pode ser necessária para a criação da pasta www, caso você receba uma mensagem de erro ao adicionar a plataforma
npm run build

// Instalação da plataforma iOS no projeto. Observe a versão. É ela que usaremos no livro
cordova platform add ios@5.0.1

// Preparação para execução no iOS
ionic cordova prepare ios
```

Para que possamos executar nossa aplicação iOS no Mac, precisamos do XCode instalado, pois é a ferramenta oficial da Apple para desenvolvimento. Se você não tiver o aplicativo em seu Mac, precisará instalá-lo pela `App Store`. No momento da escrita deste livro, ele estava na versão 10.2.01 (10E11001).

Para que você se torne um desenvolvedor certificado pela Apple e possa fazer parte do `Apple Developer Program`, existe uma série de tramitações que devem ser seguidas e que estão muito bem documentadas no portal da Apple. Entretanto, para fazer um deploy como um desenvolvedor "simples" e sem custo, existe um processo mais simples de ser realizado, e é ele que veremos aqui.

O primeiro requisito é ter um Id Apple. Se não tem, obtenha um em https://appleid.apple.com/account#!&page=create/. Em seguida, em seu Mac, acesse o XCode e acesse o `XCode Menu -> Preferences`. Na categoria `Accounts`, clique na base esquerda, no botão `+` para

adicionar seu Id Apple. Sua senha será solicitada neste momento e a janela a seguir é exibida.

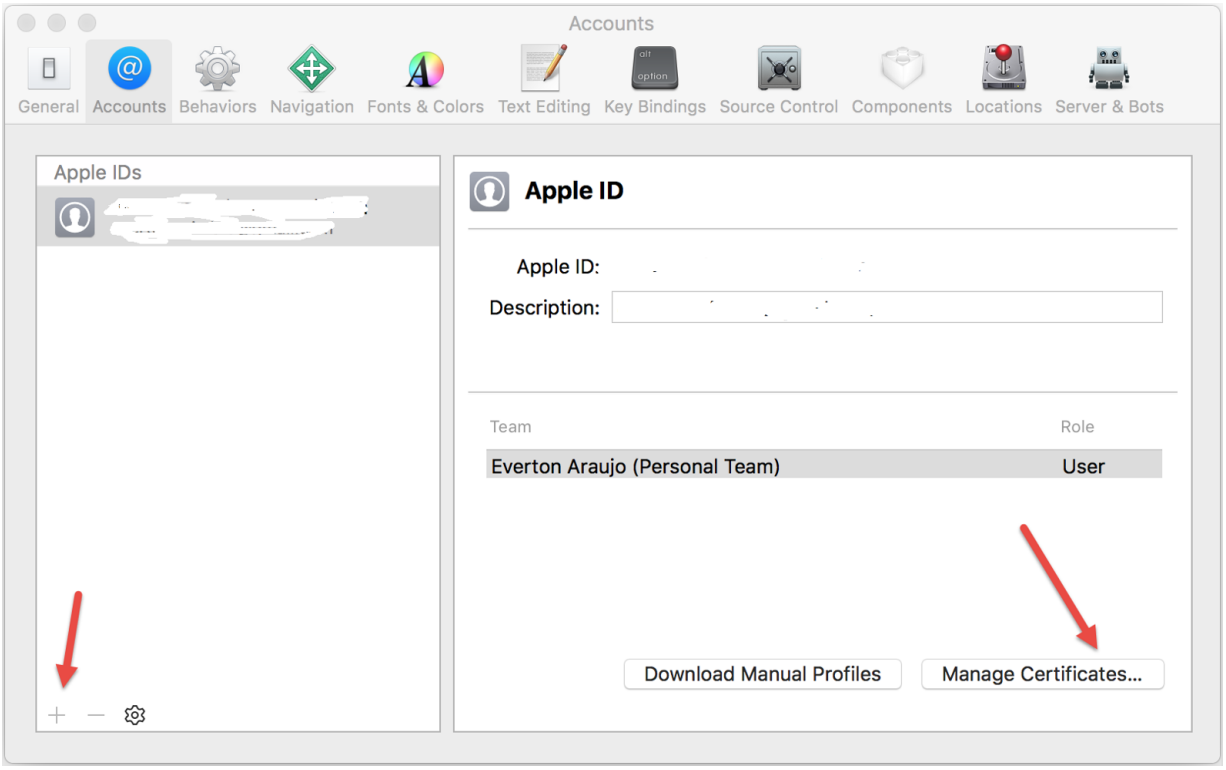


Figura 2.7: Criando uma conta no XCode

Com a conta criada, precisaremos criar uma `Signing Identity`. Clique no botão `Manage Certificates...` e a próxima janela é exibida. Clique no botão `Create` e selecione `iOS Development`.

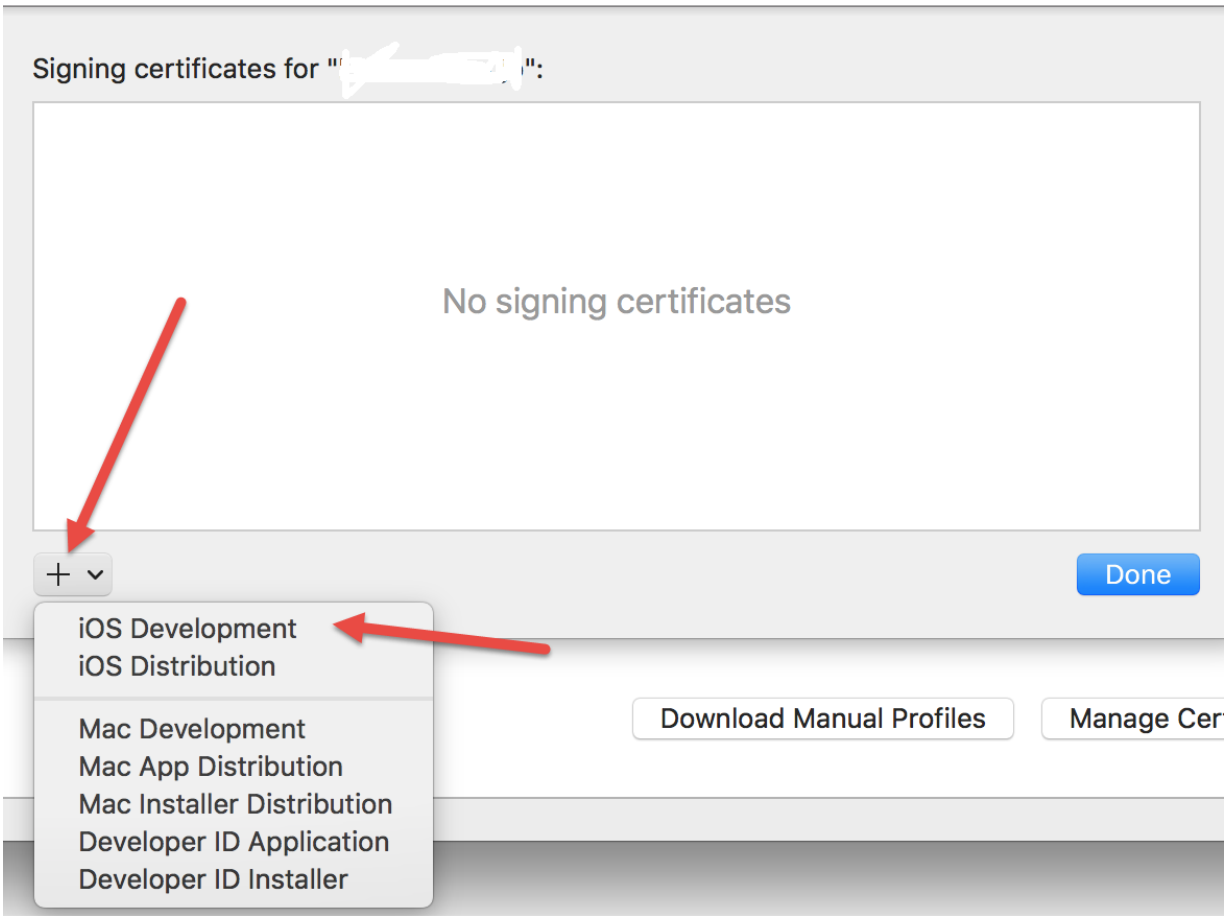


Figura 2.8: Criando uma Signing Identity

Abra o projeto gerado pelo Ionic no XCode. O que devemos fazer é abrir a pasta `platforms/ios`, que temos dentro de nossa pasta `capitulo02`. Esta pasta foi gerada pela última instrução que executamos no Ionic CLI anteriormente.

Com o projeto aberto, selecione o nó principal e na guia que se abre, verifique, na categoria `Identity`, o campo `Display Name` e `Bundle Identifier`, se estão como configuramos no VSC.

Em seguida, na categoria `Signing`, marque o checkbox `Automatically manage signing` e selecione o `Team`, que você criou. Para finalizar esta etapa, na categoria `Deployment info`, selecione a versão alvo para sua aplicação. Em meu caso, selecionei a mais recente, que era a 12.2. Com esta seleção estamos dizendo que os dispositivos que

poderão executar nossa aplicação deverão estar atualizados, no mínimo, com o iOS 12.2. Se você quiser que versões anteriores possam executar seu aplicativo, basta selecioná-la. Lembre-se de que quanto menor a versão alvo, menos recursos atuais da plataforma estarão disponíveis e quanto mais recente a versão, menos dispositivos desatualizados poderão executá-la.

Com as configurações realizadas, você pode executar sua aplicação. No topo da janela do XCode existe um botão de `play` e ao lado dele o nome da aplicação, seguido pelo dispositivo onde sua aplicação executará. A princípio, são exibidos emuladores, mas se você tiver um dispositivo conectado ao Mac, ele também aparecerá na listagem. Para começarmos, escolha um emulador e execute. O processo de deploy (implantação) ocorrerá e a aplicação executará.

Vamos ao dispositivo físico agora? Conecte-o ao Mac, selecione-o na lista de dispositivos no XCode e execute a aplicação. Se neste momento uma janela de erro aparecer em seu dispositivo, informando-o para verificar configurações de confiança, não se preocupe, isso ocorre na primeira execução de uma aplicação via o mecanismo que estamos utilizando, o de depuração. Realize a configuração indicada pelo dispositivo, que é bem simples, e execute novamente sua aplicação. Em meu iPhone eu acessei `Ajustes->Geral->Gerenciamento de Dispositivo->Selecionei meu ID Apple->Confirmar confiança` . Se tudo deu certo, a aplicação será executada em seu dispositivo. Lembre-se de encerrar a execução da aplicação, no XCode, ou no próprio dispositivo.

Se formos realizar alguma alteração no projeto, as atualizações não são aplicadas diretamente na aplicação que executamos no emulador/dispositivo. Precisamos executar novamente a instrução `ionic cordova prepare ios` e voltar ao XCode e executar novamente a aplicação. Não é preciso fechar o XCode e nem o projeto nele para isso. Logo veremos a execução da aplicação diretamente pelo terminal.

2.4 Execução de forma nativa no Android com o Cordova

Para podermos testar nossa aplicação na plataforma Android com o Cordova, temos alguns passos comuns ao iOS e outros específicos, necessários pela plataforma.

Como já temos nosso ambiente Android todo configurado em nossa máquina vamos preparar nosso projeto para poder ser executado no emulador e no dispositivo. Sendo assim, na pasta do projeto, execute as instruções a seguir. Lembre-se de que, tal qual para o iOS, a execução dependerá da internet, o que pode demorar. Pode ser que alguns destes passos você já tenha realizado nas seções anteriores, mas preferi trazê-los novamente, para facilitar os testes.

```
// Esta instrução pode ser necessária para a criação da pasta www, caso
você receba uma mensagem de erro ao adicionar a plataforma
npm run build

// Instala o Cordova no ambiente da máquina, mas se já fez anteriormente,
não precisa novamente.
npm install -g cordova

// Instala a integração de seu projeto com o Cordova, mas se já fez
anteriormente, não precisa novamente.
ionic integrations enable cordova --add

// Instalação da plataforma iOS no projeto. Observe a versão. É ela que
usaremos no livro
cordova platform add android@8.00

// Após a configuração, execute a instrução a seguir
ionic cordova prepare android
```

Agora vamos executar nosso projeto, seguindo o mesmo padrão que fizemos para a execução em iOS. Execute o Android Studio e nele abra a pasta `platforms/android` . Responda negativamente, neste momento, a qualquer solicitação de atualização de ambiente e

plugins. Após todo o processo de carga do projeto pelo IDE, na parte superior de sua janela, selecione o emulador desejado e então clique no botão de `play`.

Um problema que pode ocorrer ao abrir o projeto no Android Studio é o surgimento da mensagem a seguir, mas é simples de corrigir.

```
ERROR: The minSdk version should not be declared in the android manifest file. You can move the version from the manifest to the defaultConfig in the build.gradle file. Move minSdkVersion to build files and sync project
Affected Modules: CordovaLib, app
```

Veja que na mensagem anterior é apontado problema nos projetos `CordovaLib` e `app`, no arquivo de manifesto. Sendo assim, abra o arquivo `AndroidManifest.xml` da pasta `manifests` destes dois projetos e retire as instruções a seguir e então clique em `Try again`, que aparece no topo do editor de código.

```
<uses-sdk android:minSdkVersion="19" />

// Aqui você pode retirar apenas a parte do minSdkVersion
<uses-sdk android:minSdkVersion="19" android:targetSdkVersion="28" />
```

Se você estiver em uma rede com proxy, pode surgir também um problema com o `Gradle`, uma ferramenta utilizada pelo `Android SDK` para geração dos projetos. Foi preciso editar o arquivo `gradle.properties`, que se encontra na raiz do projeto, e inserir nele o conteúdo a seguir e, após gravar o arquivo, clicar no link `Try again`, no topo da janela de código.

```
systemProp.http.proxyHost=<ip-ou-url-sem-http-ou-https>
systemProp.http.proxyPort=<porta>
systemProp.https.proxyHost=<ip-ou-url-sem-http-ou-https>
systemProp.https.proxyPort=<porta>
```

Por fim, quando a aplicação começa a ser executada, um problema relacionado a proxy também foi apresentado com a mensagem `the connection to the server was unsuccessful`. Para resolver este problema, foi preciso adicionar a seguinte tag dentro de `<platform name="android">` no arquivo `config.xml`.

```
<preference name="loadUrlTimeoutValue" value="700000" />
```

Após a alteração foi preciso executar novamente a instrução a seguir e depois executar a aplicação no Android Studio.

```
ionic cordova prepare android
```

Para executarmos a aplicação em um dispositivo Android, precisamos habilitar as opções para o desenvolvedor (*Developer Options*). Para isso, no dispositivo, vá em `Configurações -> Sobre o dispositivo -> Informações de Software` e clique 7 vezes sobre o `Número de compilação` (*build number*). Após isso, retorne para `Configurações` e clique em `Opções do Desenvolvedor` (*Developer Options*) e habilite a opção `USB Debugging` . Se estas opções não aparecerem para seu dispositivo você precisará buscar na documentação dele como acessá-las. Normalmente, nos sites de pesquisa, digitando as palavras-chaves `android ativar modo desenvolvedor <fabricante-e-modelo-dispositivo>` já são trazidas as orientações necessárias.

Com seu dispositivo liberado para testes e devidamente conectado ao seu computador, feche o emulador e, no Android Studio, selecione seu dispositivo na lista disponibilizada ao lado do `play` e execute sua aplicação. Bem mais simples que no iOS, certo?

2.5 Execução da aplicação diretamente do Terminal

Nas seções anteriores, preparamos nosso projeto para o Cordova e para as plataformas iOS e Android. Abrimos os projetos "nativos" no XCode e Android Studio e então executamos a aplicação. Se mudássemos algo em nosso projeto, precisaríamos executar novamente a preparação e então retornar ao IDE para executar novamente. Isso é muito trabalho. Felizmente, existe o Live Reload.

O Live Reload é um recurso oferecido para que sua aplicação seja atualizada em seu dispositivo ou emulador a cada compilação bem-sucedida. Para fazermos uso deste recurso, vamos executar nossa aplicação diretamente de nosso terminal. Feche o XCode e Android Studio. Se quiser executar a aplicação nos emuladores, desconecte os dispositivos. Veja as instruções a seguir.

```
npm i -g native-run

// Execução no iOS
ionic cordova run ios -l

// Exceução no Android
ionic cordova run android -l
```

Na execução no iOS não é para ocorrer problemas, mas, caso uma mensagem de inexistência de emuladores apareça, caso este seja seu alvo, você pode abrir o XCode e abrir o emulador a partir dele, pelo menu `Xcode -> Open Developer Tool -> Simulator`. Se for executar no dispositivo real, basta tê-lo conectado ao computador e a configuração de `Sign in` realizada, como demonstrado anteriormente.

Na execução iOS, pode ocorrer de os emuladores disponibilizados pelo Cordova não estarem disponíveis em seu projeto. Para resolver isso, precisamos executar as seguintes instruções, seguidas de orientações para identificação de dispositivos disponibilizados.

```
// Navegar até a pasta do cordova para o iOS
cd platforms/ios/cordova

// Instalar os emuladors em sua versão mais atualizada
npm install ios-sim@latest

// Retornar para a pasta do projeto
cd ../../../

// Remover e instalar a plataforma iOS
cordova platform rm ios
```

```
cordova platform add ios@5.0.1

// Verificação de plataformas disponíveis para execução da aplicação
ionic cordova emulate ios --target --list
ou
cordova run ios --list --target --emulator

// Execução em uma plataforma específica
ionic cordova emulate ios --livereload -c --target="iPhone-XR, 12.2"
```

Em relação ao Android, temos algumas possibilidades mais comuns de ocorrerem e veremos elas aqui. Lembra do problema do proxy com o Gradle, que comentei há pouco, na execução do Android Studio? Pois bem, na execução da instrução anterior pelo Ionic CLI, para executarmos nosso projeto, teremos o mesmo problema. Precisaremos criar, na pasta `platform/android` de nossa aplicação `capitulo02`, o arquivo `gradle.properties` e nele inserir a mesma configuração apresentada anteriormente, e que, para facilitar, trago novamente.

```
// Arquivo gradle.properties em platform/android
systemProp.http.proxyHost=<ip-ou-url-sem-http-ou-https>
systemProp.http.proxyPort=<porta>
systemProp.https.proxyHost=<ip-ou-url-sem-http-ou-https>
systemProp.https.proxyPort=<porta>
```

Com tudo configurado, caso você esteja em um proxy, execute novamente a seguinte instrução.

```
ionic cordova run android -l
```

Tive um problema para a execução da aplicação, que antecipo aqui, caso você tenha. A mensagem `version (40) doesn't match this client (39)` foi exibida. O problema foi resolvido adicionando à variável `PATH` o caminho para a pasta `platform-tools` do Android SDK. Em meu caso, o endereço completo foi `C:\Users\evert\AppData\Local\Android\Sdk\platform-tools`. Este problema não ocorreu em outra máquina de teste.

Executando a aplicação pelo Ionic CLI, sem o Emulador estar iniciado, a mensagem `PANIC: Missing emulator engine program for 'x86' CPU.` foi exibida, impedindo o emulador de ser iniciado diretamente pelo Ionic CLI. Para resolver este problema eu registrei uma variável de ambiente chamada `ANDROID_AVD_HOME`, atribuindo o local dos emuladores a ela, em meu caso `C:\Users\Everton\.android\avd`. Também adicionei à variável de sistema (não do usuário) `PATH` o caminho para a execução do emulador, em meu caso `C:\Users\Everton\AppData\Local\Android\Sdk\emulator`. Também precisei iniciar o Visual Studio Code como Administrador.

Outro problema me ocorreu em testes em uma máquina não tão atualizada, sem placa de vídeo off-board e com o Windows 7. O erro surgiu ao tentar executar o emulador criado. A mensagem `opengles emulation failed to initialize` foi exibida. Este problema está relacionado a recursos gráficos necessários para a execução do emulador. Precisei, na janela do AVD Manager, editar as configurações do emulador. Ao abrir a janela de edição, cliquei no botão `Show Advanced Settings` e em `Emulated Performance` configurei `Graphics` para `Software - GLES 2.0`. Com isso o emulador foi executado, mas muito lento.

Um problema relacionado à API 28 do Android também pode ocorrer quando formos executar nossas aplicações e você o terá se aparecer em seu dispositivo ou emulador a mensagem `ERR_CLEARTEXT_NOT_PERMITTED`. Não entrarei em detalhes sobre este erro, mas deixo o link https://medium.com/@son.rommer/fix-cleartext-traffic-error-in-android-9-pie-2f4e9e2235e6/, caso queira entender o porquê de ele acontecer. Vamos procurar resolvê-lo.

No arquivo `config.xml` vamos inserir o código a seguir abaixo das plataformas.

```
<edit-config file="app/src/main/AndroidManifest.xml" mode="merge"
target="/manifest/application"
xmlns:android="http://schemas.android.com/apk/res/android">
```

```
  <application android:usesCleartextTraffic="true" />
</edit-config>
```

Caso esta orientação ainda não surta efeito, quando o Android abrir sua aplicação, abra no editor o arquivo `app/manifests/AndroidManifest.xml` e, ao final do elemento `<application>`, antes de ele fechar, insira `android:usesCleartextTraffic="true"`.

Se mesmo assim sua aplicação não funcionar, recomendo que mude o target para a API do Android para a 27, inserindo a instrução a seguir antes da tag `<platform>` do arquivo `<config.xml>`.

```
<preference name="android-targetSdkVersion" value="27" />
```

Isso tudo é muito chato. É um erro muito comentado na rede, nesta época de escrita do livro. Todos apontam estas soluções que passei e confesso que em alguns casos, funcionou aqui comigo, mas em outros, estranhamente, tive que apenas para baixar o nível da API. Espero que quando você esteja lendo este livro este problema não esteja mais ocorrendo.

Com todas estas possibilidades comentadas, torcendo para tudo funcionar direitinho com você, na execução da instrução anterior do Ionic CLI, depois do download de todas as dependências para execução do projeto no Android, serão realizados os processos de build (construção) e deploy (implantação) da aplicação. Na sequência, se o emulador estiver ativo ou um dispositivo esteja conectado ao computador, a aplicação será executada nele, caso contrário, o emulador é aberto.

Um registro é importante fazer, que é o tempo de espera em todos os processos de download, preparação e execução, quando se utiliza uma máquina que possui um HD SSD, o tempo de resposta é imensamente menor do que em uma máquina com HD mecânico. Ainda, no Mac, mesmo com processador inferior, todo o processo também é mais rápido que em uma máquina Windows. Estas

informações levam em conta as máquinas que utilizei para os testes.

Houve momentos em que a aplicação não era distribuída e instalada via Ionic CLI, o que me levava a fechar o emulador e abri-la e executá-la por meio do Android Studio. Depois ela retornava a funcionar. Passei vários momentos tentando entender este bug, mas diversos links apontavam a mesma solução.

Outra observação importante é relacionada ao tempo de inicialização de um emulador, principalmente Android, no Windows. Procure deixar seu emulador sempre ativo, baixando-o apenas quando não for mais precisar dele, com certeza.

2.6 O Capacitor no lugar do Cordova

O Ionic é integrado também a outro framework para execução de aplicativos Ionic de maneira híbrida, conhecido como Capacitor. Segundo alguns fóruns e artigos, além do próprio site (https://capacitor.ionicframework.com), o Capacitor é o sucessor do Cordova e muitos projetos vêm sendo desenvolvidos diretamente para o Capacitor, mesmo que utilizando plugins Cordova. Ressalto aqui que tanto o Cordova como o Capacitor, não são exclusivos para o Ionic. Vamos então preparar o ambiente para o uso do Capacitor.

iOS

Como já dito, para executarmos aplicações Ionic iOS em emuladores e dispositivos, com os recursos que temos até este momento de leitura do livro, precisamos de um Mac. A preparação de nosso ambiente para o uso do Capacitor, demanda um conjunto de pré-requisitos e dependências. A primeira delas se refere à instalação da dependência `CocoaPods` , que é um gerenciador de

dependências para projetos `Swift` e `Cocoa Objective-C`. Não entrarei em detalhes sobre isso, por não fazer parte do escopo do livro. Sendo assim, no terminal de seu Mac, digite as instruções a seguir.

```
// Instalação do Cocoapods. A versão neste momento é a 1.6.2
sudo gem install cocoapods

// Instalação Xcode Command Line tools, se necessário, mas já instalamos
quando utilizamos o Cordova
xcode-select --install

// Atualização do repositório local do CocoaPods
pod repo update
```

O processo seguinte refere-se à configuração de nosso projeto para uso do Capacitor. Algumas instruções são necessárias, e as trago na sequência. Eu optei por mostrar para você como remover as plataformas criadas pelo Cordova, mas isso não é necessário.

```
// Remoção do Cordova, caso queira
ionic cordova platform rm ios
ionic cordova platform rm android

// Desabilitar a integração com o Cordova, caso queira
ionic integrations disable cordova

// Instalar a integração de seu projeto com o Capacitor
ionic integrations enable capacitor
```

Ao término da instrução anterior, que habilita a integração com o Capacitor, precisamos abrir e realizar algumas configurações no arquivo `capacitor.config.json`, que foi criado na raiz de nosso projeto. Estas configurações se referem às propriedades `appId` e `appName`, que seguem o mesmo princípio apresentado para o projeto em Cordova. Eu utilizei as configurações a seguir. Em relação à versão instalada, no momento da escrita deste livro foi a `1.0.0`, recém-lançada.

```
"appId": "br.com.casadocodigoionic",
"appName: "Capitulo02"
```

Se por algum acaso acontecer de o arquivo anterior ser apagado, para que ele seja novamente criado, você precisa executar `npx cap init` e responder às perguntas para os dados da listagem anterior. Após a configuração, precisamos dar sequência em instruções para preparar nosso projeto. Veja-as a seguir.

```
// Adicionando o Capacitor para iOS ao projeto.
ionic capacitor add ios
```

Tal qual fizemos para o Cordova, precisamos abrir nosso projeto Capacitor no XCode, mas o Ionic CLI pode fazer isso por nós, por meio da instrução a seguir.

```
ionic capacitor open ios
```

Precisamos configurar a aplicação para os pré-requisitos da Apple, novamente, igual fizemos para o Cordova. Sendo assim, no XCode, já aberto, clique no nó `App`, no lado esquerdo da janela e nas configurações gerais da aplicação, selecione o `Team`. Verifique o `Bundle Identifier` e `Display Name`. Veja também o nome para a aplicação e seu id. Com as configurações realizadas, selecione o emulador ou dispositivo e execute sua aplicação.

E se realizarmos alterações em nosso projeto Ionic? Precisamos novamente realizar o processo de distribuição, com as instruções a seguir, sendo que a primeira já conhecemos. Não é preciso fechar e abrir novamente o XCode.

```
// Gera os arquivos necessários e copia para a pasta www
npm run build

// Copia arquivos da pasta www para a pasta ios/App/public
ionic capacitor copy ios
```

A boa notícia é que o Capacitor traz uma funcionalidade a mais para aplicações iOS, que o Cordova só tem, até o momento, para o Android, que é o `Live Reload`. Isso nos permite executar a aplicação no dispositivo e realizarmos alterações em nosso projeto e vermos estas atualizações serem realizadas no aplicativo em execução.

Para termos estas características, precisamos de alguns passos e de mais algumas configurações. A primeira é que tanto o Mac, quanto o dispositivo, estejam na mesma rede. Com esta verificação realizada, vamos executar nosso projeto pelo Ionic, com a instrução a seguir.

```
ionic serve
```

Verifique o IP informado pelo Ionic CLI para acesso externo à aplicação. Para garantir que o Mac e dispositivo estejam na mesma rede, acesse o navegador em seu dispositivo e informe a URL para a aplicação, que foi disponibilizada pelo Ionic CLI.

Vamos agora configurar nosso arquivo `capacitor.config.json`, onde precisamos associar este IP a uma propriedade de objeto chamada `server`. Veja a instrução a seguir. Lembre-se da vírgula no final da linha de `webDir`.

```
"server": {
    "url" : "http://<ipionicserver:porta>"
}
```

Muito bem, com nossa aplicação em execução, precisamos configurá-la para uso no dispositivo. Alguns passos já conhecemos. Veja as instruções a seguir. Apenas para repetir meus passos, caso queira, feche o XCode, mas não é necessário. Para a execução das instruções a seguir, abra uma nova janela do Terminal.

```
npm run build

// Configura o Capacitor para estar com a plataforma iOS sincronizada
ionic capacitor sync ios

ionic capacitor open ios
```

Com o XCode aberto, execute sua aplicação. Modifique alguma coisa no `home.page.html` (pelo Visual Studio Code), grave e observe seu dispositivo. A alteração apareceu? Legal, não é? Caso em

algum momento as alterações não sejam replicadas, será preciso parar o Ionic Serve e realizar o processo novamente.

Android

O procedimento para utilização do Capacitor para o Android é semelhante ao que vimos para iOS. Para termos a sincronização entre aplicação e desenvolvimento precisamos da mesma configuração no `capacitor.config.json` , e a execução das instruções a seguir, que já são conhecidas.

```
// Remoção do Cordova, caso queira
ionic cordova platform rm ios
ionic cordova platform rm android

// Desabilitar a integração com o Cordova, caso queira
ionic integrations disable cordova

// Instalar a integração de seu projeto com o Capacitor, caso ainda não
tenha feito
ionic integrations enable capacitor

// Configuração do capacitor.config.json

// Preparação da pasta www, caso não tenha feito ainda
npm run build

// Adição do Android para o Capacitor
ionic capacitor add android

// Se for executar sem sincronização
ionic capacitor copy android

// Se for usar com sincronização
ionic capacitor sync android

// Abrir o Android Studio para execução da aplicação
ionic capacitor open android
```

Lembre-se de que se você estiver em uma rede com proxy, precisará configurar isso no arquivo `gradle.properties` quando o Android Studio abrir. O processo de ter toda a aplicação carregada e configurada, pela primeira vez, pode demorar um pouco, pois há todo o processo de configuração do projeto que o AS faz e ele depende de diversos downloads.

A mensagem `ERR_CLEARTEXT_NOT_PERMITTED` que vimos anteriormente, utilizando o Cordova com Android, pode aparecer também na utilização do Capacitor. Lembre-se de todas as dicas que dei anteriormente ok?

Vimos nessa seção, com a apresentação do Capacitor, que temos um certo workaround para fazermos uso do sincronismo entre aplicação em execução nos dispositivos/emuladores e ambiente de desenvolvimento, mas ele é uma ferramenta que está evoluindo e o objetivo é ganhar espaço e trazer produtividade.

2.7 Alguns problemas na execução nativa das plataformas e a depuração

Esta seção é simples, curta e objetiva e terá como foco apresentar alguns problemas que podem ocorrer na preparação e execução dos projetos em plataformas específicas.

Houve alguns momentos em que ao preparar a aplicação para o Cordova/iOS, alguns erros relacionados a arquivos básicos da plataforma apareceram. E para resolver, precisei realizar alguns workarounds. O primeiro deles foi desinstalar o Cordova globalmente do ambiente, com `npm uninstall -g cordova` e em seguida realizar a instalação novamente, com `npm install -g cordova` .

Outro workaround foi remover a pasta `platforms` , `plugins` , `resources` e os arquivos `config.xml` e `capacitor.config.json` e realizar todo o

procedimento de preparação do projeto para as plataformas. Há situações em que executar o `npm install` também pode auxiliar. Infelizmente, são problemas encontrados em algumas situações durante a preparação e execução dos projetos em plataformas específicas.

Depurar uma aplicação é tão importante quanto a criação de código. Entretanto, depurar um código TypeScript, ou JavaScript, em tempo de execução não é algo trivial, pela característica das tecnologias empregadas. Vimos que temos o console do navegador, uma ferramenta poderosa. Mas e quando utilizamos um dispositivo? Também fazemos uso do console, que no caso no Chrome, é de fácil acesso, mas também há disponibilidade dele para o Safari.

Para termos o console do dispositivo acessível, utilizando o navegador Chrome, precisamos informar a URL chrome://inspect/#devices/. Os dispositivos físicos conectados a seu equipamento ou via rede aparecerão na página aberta, assim como a aplicação em execução no emulador. Precisamos então clicar no link `Inspect` que aparece e nossa janela de console se tornará disponível. Isso ajuda e muito.

Conclusão

Chegamos ao final deste segundo capítulo. Conseguimos identificar as ferramentas que utilizaremos, realizamos as instalações delas e as testamos em um projeto simples. Foi possível também executarmos a aplicação desenvolvida em dispositivos físicos e emuladores específicos do iOS e Android, por meio do Cordova e do Capacitor, o que nos viabiliza o uso nativo de recursos dos aparelhos. Para o teste mais simples, vimos o DevApp, uma aplicação da Ionic para execução de aplicativos em dispositivos, sem toda a burocracia definida pelas plataformas. Pudemos verificar que existe sempre a possibilidade de ocorrência de alguns erros/problemas, mas todos contornáveis. Muita coisa interessante foi vista, e estamos apenas no segundo capítulo. Vamos pegar um fôlego antes de começarmos o terceiro?

Capítulo 3
A primeira aplicação Ionic

No capítulo anterior, criamos uma aplicação básica, com vistas apenas a testar o ambiente necessário para desenvolvimento e execução de aplicações Ionic.

A aplicação que iniciaremos aqui será responsável pela interface com o usuário para manipulação de dados relacionados a clientes.

Começaremos o capítulo apresentando a estrutura básica de um projeto criado pelo Ionic CLI, os componentes básicos implementados no `home` e de como ocorre a execução da aplicação. Na sequência, criaremos nosso template para clientes com alguns componentes para a interação com o usuário.

3.1 A estrutura básica de um projeto Ionic

Vamos criar um novo projeto para trabalharmos neste capítulo. Desta maneira, em sua pasta de projetos Ionic, insira no terminal, como sugestão, a seguinte instrução. Observe que optei pelo não uso do GIT neste projeto.

Quanto à instalação da integração com o Cordova ou Capacitor, fica a seu critério, com base no que trabalhamos no capítulo anterior e seu desejo de testar a aplicação em emuladores ou dispositivos, por meio destes frameworks.

```
// Lembre-se de estar na pasta raiz de seus projetos
ionic start capitulo03 blank --type=angular --no-git

// Acesse a pasta criada para seu projeto e execute a instrução a seguir
npm install --save @ionic-native/core@5.7.0
```

Após a criação, vamos abrir a nova pasta (capitulo03) no Visual Studio Code. Veja a estrutura criada na figura a seguir.

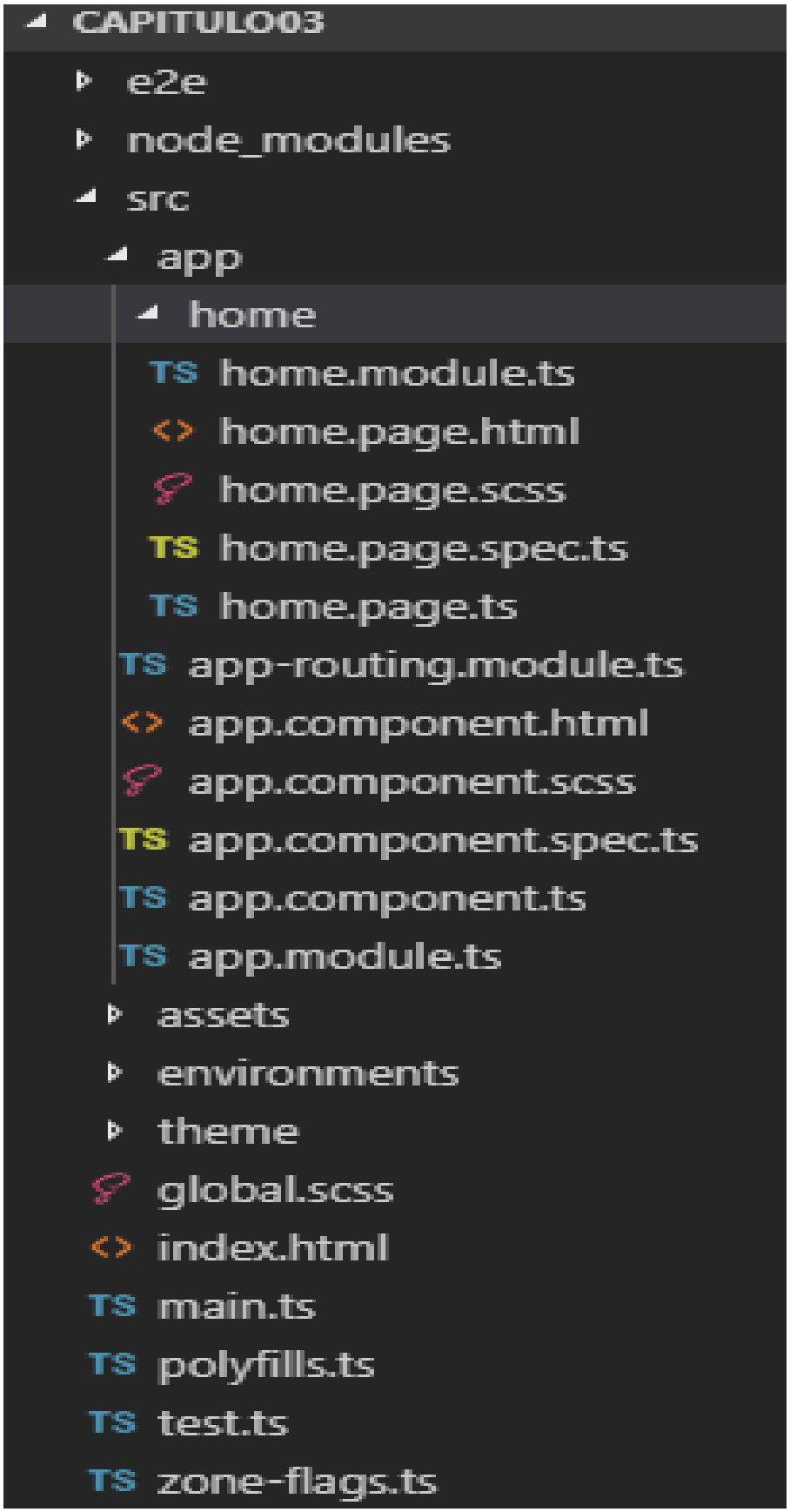

CAPITULO03
e2e
node_modules
src
app
home
home.module.ts
home.page.html
home.page.scss
home.page.spec.ts
home.page.ts
app-routing.module.ts
app.component.html
app.component.scss
app.component.spec.ts
app.component.ts
app.module.ts
assets
environments
theme
global.scss
index.html
main.ts
polyfills.ts
test.ts
zone-flags.ts

Figura 3.1: Estrutura criada para o projeto

A estrutura adotada por nossa aplicação Ionic 4 segue a estrutura básica de uma aplicação criada em Angular, que no momento da escrita deste livro, está na versão 7.2.2 e foi o tipo de framework para templates que escolhemos. Note que temos a pasta `src` na raiz da pasta do projeto. Dentro dela, temos outra, chamada `app`, que é onde ficará o código que implementaremos para nossa aplicação. A fala sobre Angular em alguns pontos do livro terá o foco de apenas demonstrar a relação entre os frameworks. Você não precisará aprender o Angular para nossas atividades, mas é uma ferramenta cujo estudo recomendo.

O Angular tem a convenção de organizar seus artefatos (arquivos) em pastas e, dentro delas, que podemos chamar de contextos, por enquanto, os arquivos que são relacionados a elas. A este contexto, o Ionic dá o nome de componente ou módulo, dependendo de como os artefatos forem criados. Veremos isso quando começarmos a trabalhar nos códigos. Estas pastas, de componentes, ficam sempre dentro de `src/app`, como é o caso de `home`.

Abra a pasta `home` e veja que temos 5 arquivos, todos com o nome composto por `<nome-contexto/pasta/componente>.<tipo-do-artefato>.<tipo-do-arquivo>`. Observe que destes arquivos, 4 têm o tipo de artefato `page`, pois são eles que compõem uma página, cada um com sua finalidade e tipo de arquivo. Temos:

- `html`, que se refere ao template (modelo) que será renderizado pela aplicação;
- `scss` são arquivos que contêm os CSSs relacionados ao template em questão. SCSS é um arquivo que contém código Sass, que é uma linguagem com recursos para manipulação de estilos;
- `ts` conterá código TypeScript, que serão `transpilados` (traduzidos) para JavaScript; e
- `spec.ts`, que mantém códigos de testes para o componente em questão. Nós não trabalharemos com testes neste livro.

Abra o arquivo `home.page.html` , clicando no nome dele, pois é assim que o Visual Studio Code abre um arquivo para edição. Trago para discutirmos o código existente neste arquivo, que já conhecemos do capítulo anterior.

```
<ion-header>
  <ion-toolbar>
    <ion-title>
      Ionic Blank
    </ion-title>
  </ion-toolbar>
</ion-header>

<ion-content padding>
  The world is your oyster.
  <p>If you get lost, the <a target="_blank" rel="noopener"
href="https://ionicframework.com/docs">docs</a> will be your guide.</p>
</ion-content>
```

Todo o código que temos são tags que representam componentes do Ionic e, se necessário, as HTML. O arquivo HTML, apresentado na listagem, é chamado, pela convenção do Ionic/Angular, de template/modelo. Podemos ainda criar nossos componentes e utilizar componentes de terceiros na criação de nossos templates.

3.2 Os componentes básicos do template home

No VSC, ative um terminal, clicando no menu `Terminal -> New Terminal` . Existe um conjunto de teclas de atalho para isso, que você pode utilizar, que é `CTRL + SHIFT + '` .

Por praticidade, realizarei as execuções com mais frequência no próprio navegador. Como estamos começando no livro, vou trazer novamente aqui a instrução que realizará esta execução. Veja na sequência o código e após ele e os componentes identificados na imagem.

```
ionic serve --lab
```

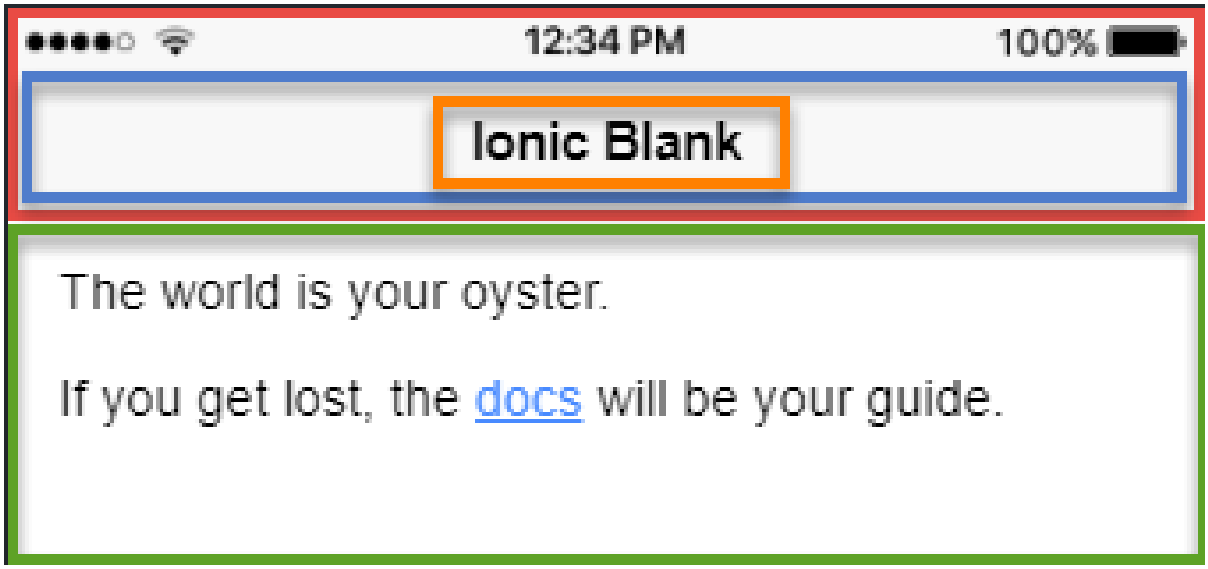


Figura 3.2: Identificando os componentes no template em execução

Observe o retângulo vermelho (o mais externo), ele representa o `<ion-header>` e é um componente obrigatório para todos os templates de página. Dentro deste componente, temos o `<ion-toolbar>`, representado pelo retângulo azul. Tente escrever algo dentro dele, antes do `<ion-title>`, e verifique a aplicação em execução. Ele escreve o que digitamos, mas sem respeitar as características das plataformas para o título da página. Por isso, fazemos uso do componente `<ion-title>` dentro da toolbar, pois, na hora de executar a aplicação, o Ionic respeitará cada plataforma. Na sua execução, veja que para o Android o título é alinhado à esquerda e não ao centro, como no caso do iOS. Todo o restante de nossa página, representado pelo retângulo verde, é a área de conteúdo e que é identificada pelo componente `<ion-content>`. A diretiva `padding` informa que será preciso realizar um espaçamento entre as bordas do template e seu conteúdo. A título de curiosidade, remova a diretiva e veja a diferença na execução.

Para o Ionic, a definição do template, pelo arquivo HTML, não é suficiente para que a página seja renderizada. Ela é um artefato, que utiliza outros artefatos. Então, ela precisa de informações de

como estes artefatos, e ela, serão renderizados. Veja o código de `home.page.ts` na sequência.

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-home',
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss'],
})
export class HomePage {

}
```

Observe que temos uma classe chamada `HomePage`, que é precedida por um `decorator`, chamado `@Component`. Este decorator receberá um objeto JavaScript, com três propriedades: a `selector` define como o componente será utilizado/referenciado na aplicação. Como nosso componente tem escopo de página, esta propriedade poderia não ser configurada aqui; a `templateURL` aponta para o modelo que será utilizado para renderizar a classe. Veja que ela aponta para a página que discutimos há pouco; por fim, temos a propriedade `styleUrls`, que também define qual o arquivo, exclusivo ao template, que conterá código CSS que será utilizado para o template em questão. Decorators marcam elementos de código, neste caso, o `@Component()` precede uma classe, chamada `HomePage`.

Outro ponto importante é a instrução `export` antes da definição da classe. É desta maneira que definimos que a classe poderá ser consumida por outros módulos e componentes.

Para que nosso componente `Home` esteja disponível para a aplicação, precisamos empacotá-lo em um módulo. Abra o arquivo `home.module.ts`. Para facilitar, na sequência trago este código.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { IonicModule } from '@ionic/angular';
import { FormsModule } from '@angular/forms';
```

```
import { RouterModule } from '@angular/router';

import { HomePage } from './home.page';

@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    IonicModule,
    RouterModule.forChild([
      {
        path: '',
        component: HomePage
      }
    ])
  ],
  declarations: [HomePage]
})
export class HomePageModule {}
```

Os imports trazem, para o contexto do artefato em questão, os módulos, plugins, componentes e qualquer tipo de artefato que seja necessário para a classe (ou módulo). Observe que temos a definição da classe precedida de um `decorator` do Angular, o `@NgModule()`, que recebe um objeto JavaScript.

O `RouterModule` é o responsável pela navegação entre páginas. Veja que ele invoca o método `forChild()`, que é recomendado para carregamento tardio (*lazy loading*) de submódulos, ou seja, o módulo será carregado apenas quando for requisitado. Veja que o argumento para este método é um array de objetos JavaScript, que definem uma rota. A propriedade `path` refere-se ao caminho que define quando o módulo será requisitado e `component`, que define qual componente será retornado quando a rota definida em `path` for informada. Em nosso exemplo, quando a raiz de nossa aplicação for requisitada, o componente `HomePage` será renderizado.

Note a propriedade `declarations` do `decorator`, que informa quais componentes/classes são declarados no módulo em questão.

Informamos o `HomePage`, entre colchetes, o que caracteriza um array.

3.3 A execução da aplicação

Com a explanação realizada para os componentes da página `home`, que é criada para todo projeto do modelo `blank`, podemos agora buscar uma compreensão de como ocorre a execução da aplicação.

Abra o arquivo `index.html`, que está na raiz da pasta `src`. No final do arquivo, dentro da tag `<body>`, note a existência de um componente Ionic, chamado `<app-root>`.

O `<app-root>` é o componente de inicialização de uma aplicação Ionic. Abra o arquivo `main.ts`, que está também na pasta `src`. Observe as instruções a seguir, que estão no final do arquivo. Veja que é enviado `AppModule` para o método `bootstrapModule()`, encadeado ao `platformBrowserDynamic()`. Esses métodos inicializam a aplicação e indicam que ela começa com o módulo `AppModule`.

```
 platformBrowserDynamic().bootstrapModule(AppModule).catch(err =>
console.log(err));
```

A maioria dos componentes do Ionic pertence a módulos, e um módulo pode conter um ou vários componentes. Os módulos podem ser vistos também como uma maneira de organizar os componentes.

Quando nossa aplicação for executada, já sabemos que ela buscará pelo arquivo `index.html` e que ele tem o componente `<app-root>`, que é o que inicializa a aplicação Ionic. Ele estará disponível para a página e aplicação por meio do código anterior e, no início do código, temos uma instrução que traz para o contexto de `main.ts` o módulo `AppModule`, que é:

```
import { AppModule } from './app/app.module';
```

É possível também verificar, no início do arquivo, vários outros imports, todos necessários para a inicialização da aplicação.

Mas e onde está o `AppModule` ?. Observe que, no código anterior, importamos `./app/app.module` para uma referência chamada `AppModule` . Vamos então abrir este arquivo, dentro de `src/app` . Veja no código dele `declarations` , com `AppComponent` , e `bootstrap` , que aponta o `AppComponent` , que será o componente de inicialização do módulo.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { RouterModule, RouteReuseStrategy, Routes } from
'@angular/router';

import { IonicModule, IonicRouteStrategy } from '@ionic/angular';
import { SplashScreen } from '@ionic-native/splash-screen/ngx';
import { StatusBar } from '@ionic-native/status-bar/ngx';

import { AppComponent } from './app.component';
import { AppRoutingModule } from './app-routing.module';

@NgModule({
  declarations: [AppComponent],
  entryComponents: [],
  imports: [BrowserModule, IonicModule.forRoot(), AppRoutingModule],
  providers: [
    StatusBar,
    SplashScreen,
    { provide: RouteReuseStrategy, useClass: IonicRouteStrategy }
  ],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Mas vamos então ao componente `AppComponent` . Abra o arquivo `app.component.html` . Viu que não é `page` , como no caso do `home` ? Isso nos leva a deduzir que o arquivo é um template para um componente (uma tag) que será utilizada em outro componente ou

página, como de fato vimos no arquivo `index.html`. Veja o código do template na sequência.

```
<ion-app>
  <ion-router-outlet></ion-router-outlet>
</ion-app>
```

No template, note que temos a utilização de um novo componente, o `<ion-app>` e, dentro dele, o `<ion-router-outlet>`, que veremos mais à frente no livro, mas para o momento precisamos saber apenas que é neste componente que as páginas serão renderizadas. O `<ion-app>` é o componente que define realmente que começamos com uma aplicação Ionic. Vamos abrir agora o arquivo `app.component.ts`. Logo no início, repare no decorator `@Component()`. Viu o valor de `selector`? É o `app-root` que vimos no `index.html`. Viu só toda a dinâmica e burocracia? O código da classe está a seguir.

```
import { Component } from '@angular/core';

import { Platform } from '@ionic/angular';
import { SplashScreen } from '@ionic-native/splash-screen/ngx';
import { StatusBar } from '@ionic-native/status-bar/ngx';

@Component({
  selector: 'app-root',
  templateUrl: 'app.component.html'
})
export class AppComponent {
  constructor(
    private platform: Platform,
    private splashScreen: SplashScreen,
    private statusBar: StatusBar
  ) {
    this.initializeApp();
  }

  initializeApp() {
    this.platform.ready().then(() => {
      this.statusBar.styleDefault();
      this.splashScreen.hide();
```

```
    });
  }
}
```

No corpo da classe anterior, veja o construtor, que recebe três argumentos. O Ionic trabalha com injeção de dependência por meio do construtor. Neste caso, o framework está injetando e disponibilizando para a classe os três objetos recebidos pelo construtor. Observe a definição do método `initializeApp()`, que é invocado no construtor.

Para fecharmos a seção vamos à uma breve indicação de todo o processo:

1. Executamos a aplicação;
2. O arquivo `index.html` é invocado em conjunto com a classe `main.ts`;
3. O `AppModule` é carregado, trazendo o `AppComponent` para a aplicação, que define o `<app-root>` e inicializa a aplicação Ionic com o `<ion-app>`;
4. Quando a aplicação está iniciada, a URL padrão para ela é apenas o endereço do servidor, ou seja, o `path` raiz definido no `home.module.ts` e a página `home.page.html` é renderizada em conjunto com a classe definida em `home.page.ts`.

3.4 Criação da página de clientes

Vamos dar início à implementação de nossa aplicação implementando uma página que represente a entrada de dados de nosso cliente. Uma página (page) é um componente e o Ionic CLI nos oferece um mecanismo para nos auxiliar na geração de componentes. Você pode, no terminal, digitar uma instrução e informar os dados que auxiliarão o Ionic CLI na criação do componente. Ou, pode informar diretamente na linha de comando que deseja criar uma página. Veja estas instruções na sequência.

```
// Criação com pergunta/resposta dos componentes
ionic generate

// Criação automática de nossa página
ionic generate page cliente/cliente-add-edit
```

Na segunda instrução, estamos informando que queremos criar uma página chamada `cliente-add-edit`, que deverá estar em uma pasta chamada `cliente/cliente-add-edit`.

Após a execução da instrução anterior, verifique as pastas criadas dentro de `src/app/pages`. Vamos agora codificar nosso template de componentes Ionic, o arquivo `cliente-add-edit-page.html`. Veja na sequência o código inicial para o template.

```
<ion-header>
  <ion-toolbar>
    <ion-title>
      Registro de CLIENTES
    </ion-title>
  </ion-toolbar>
</ion-header>

<ion-content padding>
  <ion-input type="text" placeholder="Nome"></ion-input>
  <ion-input type="email" placeholder="Email"></ion-input>
  <ion-input type="tel" placeholder="Telefone"></ion-input>
  <ion-input type="number" placeholder="Renda" text-right></ion-input>
  <ion-input type="text" placeholder="Nascimento"></ion-input>
  <ion-button shape="round" color="primary" expand="block" padding>
      Confirmar
  </ion-button>
</ion-content>
```

Podemos ver, dentro de `<ion-content>`, diversas tags `<ion-input>`, que é a responsável pelo componente de entrada de dados em forma de texto e um `<ion-button>`, que renderizará um componente de botão.

Observe o atributo `type` e suas variações de valores para ela, de acordo com o dado que estaremos solicitando ao usuário. Veja o uso de `text`, `email`, `tel` e `number`, que são, respectivamente: texto, email, telefone e números. Este atributo está diretamente relacionado ao tipo de teclado virtual que será exibido ao usuário quando estiver nos campos de entrada. Observe também o atributo `placeholder`, com um texto que será exibido na caixa de entrada, para auxiliar o usuário em relação ao dado que é preciso informar. Note a diretiva `text-right` para renda, para que o texto dentro do componente de entrada seja alinhado à direita.

O `<ion-button>`, em nosso exemplo, faz uso dos atributos `shape`, `color`, `expand` e da diretiva `padding`. O primeiro define a forma de renderização do botão, que está como `round` (cantos arredondados), sendo possível ainda o uso dos valores `full` e `block`. Para `color`, estamos fazendo uso de valores pré-criados e que podem ser vistos no arquivo `variables.css`, dentro da pasta `theme`, na raiz do projeto (src). O atributo `expand` define o padrão de preenchimento do componente na página, que em nosso caso ocupará todo o bloco disponível. Finalizamos o componente informando que deve ser aplicado o `padding` em todas as bordas.

Precisamos agora fazer com que nossa página seja renderizada, pois ao executarmos o `ionic serve --lab`, veremos a página `home`. Vamos então abrir o arquivo `app-routing.module.ts`, que está dentro da pasta `src/app`. Para o momento, vamos alterar a rota para o caminho `''`, que está redirecionando a requisição para `home`. Altere então para que a rota fique de acordo com o código a seguir.

```
{ path: '', redirectTo: 'cliente-add-edit', pathMatch: 'full' }
```

Se você estiver com o `lab` aberto, recarregue a página. Caso ela não renderize corretamente, será preciso interromper o `serve` e invocá-lo novamente. A figura a seguir apresenta o código anterior renderizado nos pseudoemuladores.

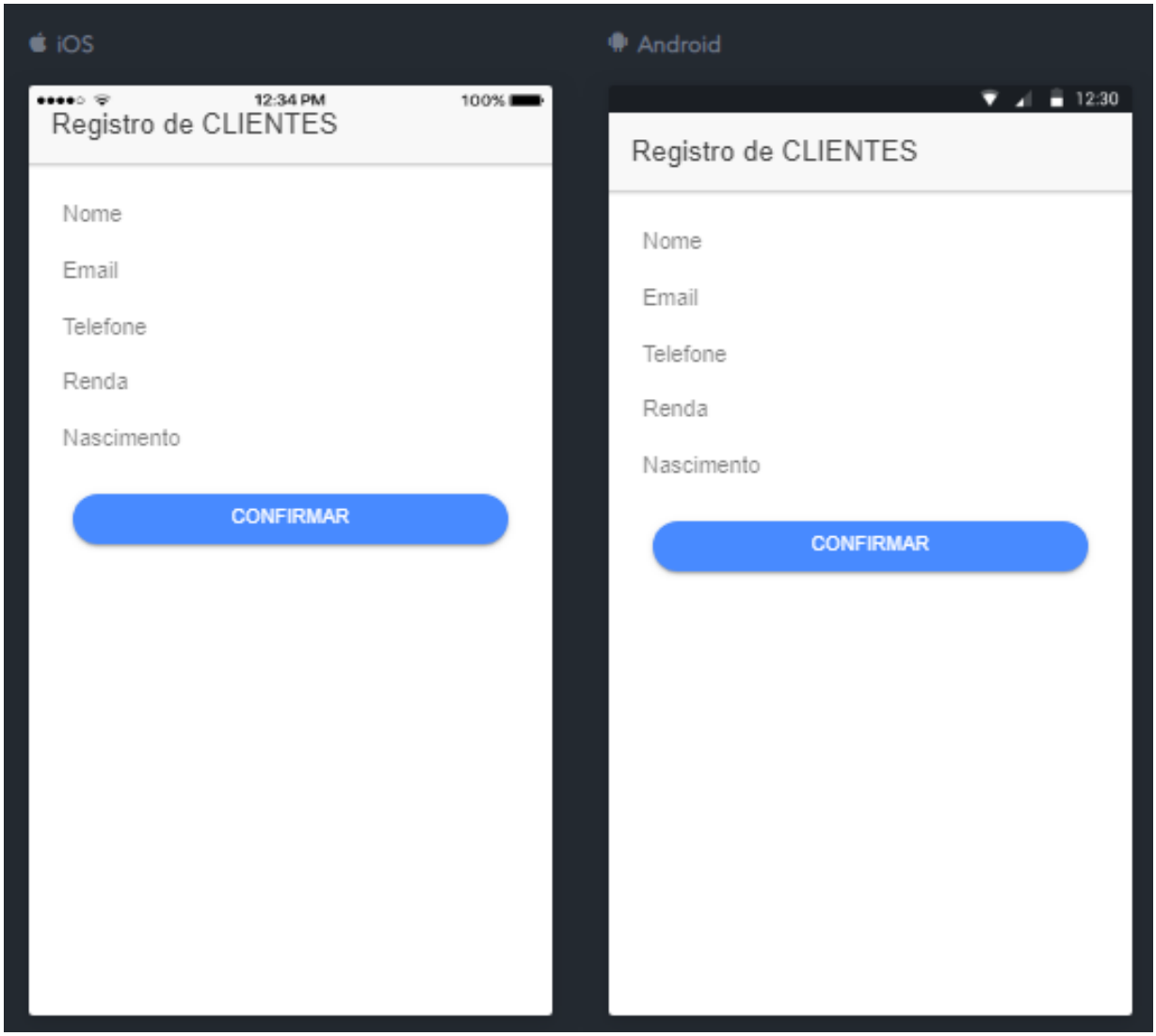


Figura 3.3: Entrada de dados de clientes

O que acha de realizar alguns testes? Comece tirando o `placeholder` dos inputs. Depois, no botão, mude os valores para `shape`, teste novas cores, retire o `expand` e, por fim, teste sem o `padding`. O que achou?

3.5 Mudanças na interface com o usuário

A maneira como implementamos os controles de entrada na seção anterior é uma forma de deixar a tela mais limpa, sem um excesso de controles, mas, se você começar a digitar os dados nos controles, perde-se a "dica" do que o controle representa. Uma saída para isso é utilizarmos controles de texto, os conhecidos *labels*. Vamos então adaptar nosso código para o que está representado na listagem a seguir. Vou apresentar apenas um controle, os demais você complementa, OK?

```
<ion-label position="stacked">Nome</ion-label>
<ion-input type="text" placeholder="Informe o nome"></ion-input>
```

Veja a execução do código anterior no navegador. Observou que o `<ion-label>` com o `position="stacked"` deixa o texto do label acima do input? Você pode ficar à vontade e mudar todos os outros, mas pode seguir a leitura e ver novas sugestões e, ao final, definir o que é melhor para sua aplicação. Vamos a uma nova possibilidade? Veja o código a seguir.

```
<ion-item no-padding>
    <ion-label position="floating">Nome</ion-label>
    <ion-input type="text"></ion-input>
</ion-item>
```

Veja na listagem anterior que agora estamos envolvendo os controles de interação em um componente `<ion-item>`, que normalmente serve para definição de itens comuns em uma renderização. Em nosso caso, temos a situação de que, para cada campo de entrada, precisamos de um `<ion-label>` e um `<ion-input>`. O `<ion-item>`, por padrão, apresenta uma linha na base, após a exibição dos componentes que ele mantém. Caso você queira ocultar esta linha, insira, no `<ion-item>`, o atributo `lines="none"`. O padrão é `full`.

Observe, no `<ion-item>`, o uso da diretiva `no-padding`. Se não a utilizarmos, o componente terá um espaço a mais nas quatro bordas. Para o momento, isso não é interessante, mas recomendo testar essa situação.

Outra mudança, com um apelo visual legal, é a `position="floating"`. Ela configura o label para que, inicialmente, seja exibido na posição do input. Entretanto, quando o foco vai para o input, o label se move para cima, simulando o comportamento do `stacked`. Devido ao uso deste recurso, retiramos o `placeholder`.

Bem, e se quisermos que uma borda seja exibida nos inputs? O padrão atual normalmente utilizado em aplicações web e móveis não é este, mas, para efeito de curiosidade e demonstração da facilidade de integração do Ionic com CSS, vamos trabalhar isso.

Para o teste de estilo, eu tirei o `padding` de `<ion-content>` e inseri `padding-top`, para um espaçamento apenas no topo do conteúdo. Meus componentes estão configurados como o código a seguir. A ideia de trocar o padding é pelo fato de que o `<ion-item>` também tem seu padding e, com isso, nossos componentes ficariam cada vez mais distantes da margem. Poderíamos utilizar o `no-padding` no `<ion-item>`, mas isso não retiraria o padding da margem direita.

```
<ion-item lines="none">
    <ion-label position="floating">Nome:</ion-label>
    <ion-input type="text"></ion-input>
</ion-item>
```

Agora, vamos aplicar alguns estilos para configuração de nosso componente input para o momento em que ele receber o foco. Desta maneira, no arquivo `cliente-add-edit.page.scss`, insira os seguintes estilos.

```
ion-input.has-focus {
    background-color: rgb(250, 248, 246);
    border-radius: 10%;
    border: 1px solid black;
}
```

O estilo que estamos definindo no código anterior é para a classe CSS `has-focus`, quando estiver sendo utilizada em um componente `ion-input`. Nosso objetivo com esta configuração está em alterar a formatação do input, para ter uma cor de fundo personalizada, ter a

borda arredondada, com espessura de 1px, sólida e preta. A ideia aqui é apresentar como configurarmos nossos componentes, buscando uma especialização em suas características visuais.

Uma dúvida que pode pairar é em como conseguimos identificar a classe CSS que devemos manipular. A resposta está na investigação dos elementos renderizados pelo navegador. O Chrome e outros navegadores trazem ferramentas que auxiliam o desenvolvedor e, como estamos utilizando o Chrome neste livro, é que utilizarei para demonstrar o uso.

Com a aplicação em execução no navegador, clique no controle de entrada para o nome do cliente. Com o controle selecionado, clique com o botão direito do mouse sobre ele e escolha a opção `Inspecionar`. Você verá a janela se dividir em duas partes. Uma com sua aplicação em execução e outra com o código renderizado e em execução no navegador, tal qual é exibido na figura a seguir.

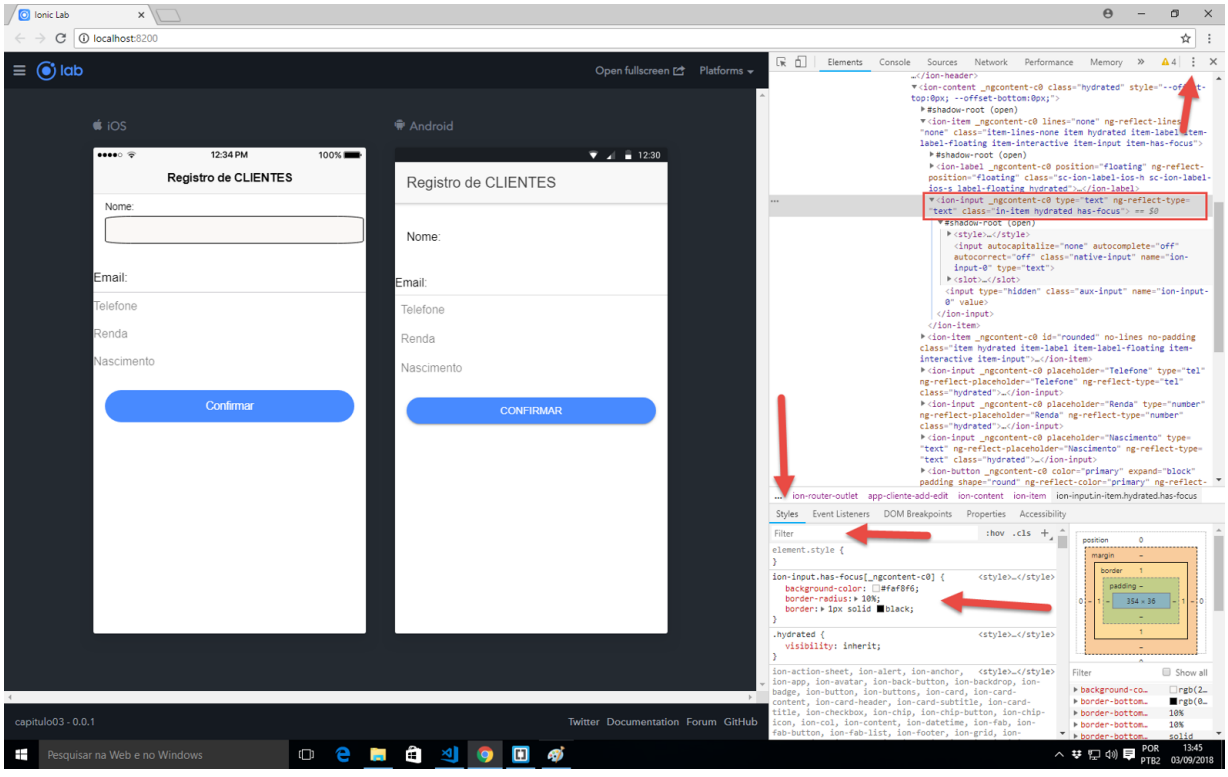


Figura 3.4: Visualização de recursos para desenvolvedor no Chrome

Você pode alterar a disposição das informações exclusivas ao desenvolvedor, clicando no botão em destaque na figura anterior e então escolhendo a posição de "dockagem" para estes recursos.

Vamos ver como identificar o código que nos interessa. Clique no controle que deseja inspecionar na aplicação e observe um efeito especial na área de código, que apontará quais os trechos de código representam o elemento selecionado. É neles que devemos buscar pelas classes que queremos especializar.

Precisamos agora realizar uma formatação em relação à margem direita de nosso input, que está relacionada ao padding do `<ion-item>` . Veja o CSS na sequência, que deve ser inserido no mesmo arquivo de SCSS que inserimos o estilo anterior.

```
ion-item {
    --inner-padding-end: 3px;
    --ion-text-color: blue;
}
```

O Ionic traz uma série de variáveis pré-configuradas, que ele utiliza e que podemos personalizar em nossos arquivos de estilos ligados a nossas páginas, ou, quando for o caso, no arquivo `variables.scss` dentro da pasta `theme` , na raiz da aplicação (src). Se você verificar no arquivo comentado, verá uma série de variáveis configuradas em um contexto chamado `:root {}` . As variáveis criadas ou alteradas neste contexto serão válidas para toda a aplicação. Podemos usar o `:root {}` em qualquer arquivo SCSS, mas é preciso ponderar bem esse uso, pois refletirá em toda a aplicação.

Voltemos a nosso estilo anterior, ligado ao componente `<ion-item>` . Estamos alterando o valor das variáveis `--inner-padding-end` para `3px` e `ion-text-color` para `blue` . Estas alterações serão válidas apenas para os componentes do tipo declarado (`<ion-item>`) no contexto da página em que estamos trabalhando. Mas como descobrir quais variáveis temos e quais devemos utilizar? O primeiro passo, com certeza, é a documentação do Ionic. Outro, mais empírico, é fazermos uso das ferramentas para o desenvolvedor do

navegador e irmos alterando as variáveis disponibilizadas para vermos o comportamento delas na aplicação. Quando identificarmos o comportamento desejado, significa que encontramos a variável. A área onde devemos realizar a busca de variáveis também está em destaque na figura anterior. Veja o resultado visual das mudanças na execução de sua aplicação.

Ao alterar o `--ion-text-color` estamos mudando a cor de todo texto exibido em todos `<ion-item>` e seus componentes filhos, que fizerem uso dessa variável, inclusive o que pertencer a outras tags HTML. Mas pode ser que o desejo seja mudar apenas os que pertençam diretamente ao componente. Poderíamos então utilizar o estilo a seguir. Observe que atribuímos a `color` o valor (`var`), de uma variável (`--ion-color-primary`).

```
ion-item {
    --inner-padding-end: 3px;
    color: var(--ion-color-primary);
}
```

O Ionic 4 faz uso de CSS4, o que permite o uso de variáveis para definição de estilos. Uma grande vantagem é que os valores dessas variáveis podem ser alterados em tempo de execução, diretamente no navegador.

Vamos agora alterar a cor do texto a ser digitado nos componentes `input` , adicionando mais um estilo, agora para o `<ion-input>` . Veja o código na sequência e teste sua aplicação.

```
ion-input {
    color: black;
}
```

3.6 Recuperação dos dados informados

Vamos trabalhar agora a recuperação de dados informados pelo usuário. A primeira técnica que utilizaremos é relacionada a variáveis de template (*template variables*). Neste caso, atribuímos um nome a um componente, que pode ser utilizado em qualquer componente do template e também em nosso código TypeScript. Veja o código a seguir.

```
<ion-input #inputNome type="text"></ion-input>
<ion-input ref-inputEmail type="email"></ion-input>
```

Observe a inserção de `#inputNome` e `ref-inputEmail`. É desta maneira que declaramos uma variável de template. Podemos usar o símbolo `#` antes do nome para a variável, ou utilizar o prefixo `ref-`.

Mas como enviarmos para a aplicação os valores da camada de apresentação? Vamos adaptar nosso `<ion-button>`, tal qual segue o código.

```
<ion-button shape="round" color="primary" expand="block" padding
 (click)="submit(inputNome, inputEmail)">
```

Observe no código anterior o trecho `(click)="submit(inputNome, inputEmail)"`. Estamos realizando uma operação de binding de evento (*event binding*), onde associamos ao evento `click` do botão, a invocação a método `submit()`, que receberá dois argumentos, que são os objetos representados pelas variáveis. Fique atento aos parênteses que guardam o nome do evento.

Precisamos implementar este método em nossa classe do componente, o arquivo `cliente-add-edit.page.ts`. Veja na sequência o método `submit()` implementado dentro da classe `ClienteAddEditPage`.

```
submit(inputNome, inputEmail) {
    console.log(inputNome.value + ' / ' + inputEmail.value);
}
```

Na assinatura do método anterior, não estamos tipificando os argumentos recebidos, o que, pelo padrão do TypeScript, define `any` como seus tipos. Ou seja, qualquer tipo de dado. Como sabemos

que os valores que chegam no método são componentes `input`, invocamos a propriedade `value` deles, para exibir na console do navegador, os valores informados pelo usuário.

Execute sua aplicação, acesse a área do desenvolvedor, pressionando `F12` ou clicando com o botão direito e então na opção `Inspecionar`. Identifique o menu `Console` na janela que se abre. Digite os valores para nome e email e clique no botão. Veja que os valores digitados serão exibidos.

Poderíamos mudar a invocação do método no HTML do template, para já enviar o valor e não o componente. Poderíamos também agora tipificar nossos argumentos no método. Veja o novo código na sequência.

```
(click) = "submit(inputNome.value, inputEmail.value)"

submit(inputNome: string, inputEmail: string) {
    console.log(inputNome + ' / ' + inputEmail);
}
```

O Angular/Ionic, oferece outros mecanismos, melhores e mais intuitivos para este processo que vimos, que são conhecidos como ligação (binding). Veja o código a seguir.

```
@ViewChild('inputNome', {read: ElementRef}) nome: ElementRef;
```

Temos a definição de uma propriedade, no escopo da classe, chamada `nome`, tipificada como `ElementRef`. A propriedade está "decorada" com `@ViewChild()`. O Visual Studio Code apontará erros de compilação, pois desconhece `ViewChild` e `ElementRef`. Precisamos importar estes elementos para a classe do componente. Se você clicar nas instruções, o VSC exibirá uma lâmpada e, ao clicar nela, ele oferecerá a possibilidade de importar o que for necessário para corrigir o problema. Ao final, teremos um `import` atualizado, conforme podemos ver na sequência.

```
import { Component, OnInit, ViewChild, ElementRef } from '@angular/core';
```

Como pudemos ver, o `@ViewChild` é um decorator de propriedades. Ele realiza a busca no DOM (Document Object Model) da página pelo componente enviado como primeiro argumento. Caso você não saiba ainda o que é DOM, ele é, de maneira simplificada, uma estrutura com todos os elementos (tags) que são utilizados para a renderização de uma página (documento) HTML. Já o `ElementRef` permite o acesso direto a um componente do DOM. Em nosso exemplo, o `@ViewChild` buscará o componente e entregará uma referência para ele na propriedade `nome`, que é do tipo `ElementRef`. Neste caso, `nome` poderia ser qualquer elemento HTML.

E o `{read: ElementRef}`, o que é? Ocorre que o `@ViewChild()` buscará e retornará o elemento nomeado, que será um componente `<ion-input>` e não um componente HTML, que é o que queremos. Veja o código a seguir para o método `submit()`. Observe que invocamos `value`, que pertence a `nativeElement`. Se não utilizarmos o `read`, não teremos estas propriedades. Recomendo que tente exibir apenas a variável `nome` na console, para comprovar esta informação. Lembre-se de tirar os parâmetros na invocação de `submit()` no `<ion-button>`.

```
submit() {
    console.log(this.nome.nativeElement.value);
}
```

3.7 Uso de binding de componentes e formulários

Quando desenvolvemos uma aplicação com uma interface gráfica com o usuário, trabalhamos para que os dados informados nesta interface sejam facilmente obtidos pela classe que implementa a visão. Poderíamos pensar nessa nossa classe como a camada de controle de uma aplicação. O Ionic/Angular oferece recursos para binding (ligação) de componentes visuais com propriedades de nossa classe. Nós já vimos ligação um pouco antes, mas foi uma

ligação de evento, com `(click)`. O que queremos agora é o `data binding`, uma ligação com nosso modelo de dados. Veja a implementação a seguir.

```
<ion-input [(ngModel)]="nome" type="text"></ion-input>
```

Observe o `[(ngModel)]="nome"`. Com essa codificação, estamos orientando o Ionic para que atribua à propriedade `nome` o valor informado no componente `<ion-input>` e, caso o valor da propriedade `nome` seja alterado no código, este valor será levado ao componente visual. Se utilizarmos apenas os `[]`, a propriedade não receberá o valor, mas o valor atribuído a ela será retornado ao componente. O que acha de testar isso? Veja a seguir o código para o método `submit()`. Retire os parâmetros também no `<ion-button>`.

```
// Nova declaração para nome
private nome: string;

submit() {
    console.log(this.nome);
    this.nome = 'Atronomogildo';
}
```

O uso de `()` garante que o valor seja alterado na propriedade, pois ele invoca o evento `change` do componente. Tente implementar a seguinte instrução, mantendo o mesmo código para `submit()`. Teste e verifique que agora o modelo é atualizado com o dado do componente, mas a instrução anterior é mais curta e resolve a situação.

```
<ion-input [ngModel]="nome" (ngModelChange)="nome = $event" type="text">
</ion-input>
```

Voltando à premissa de que uma aplicação Ionic segue o modelo de aplicações web, precisamos reforçar que o HTML faz uso de formulários para os controles de entrada de dados por parte do usuário e consequente submissão destes dados para uma aplicação servidora. No Ionic, esta aplicação servidora por enquanto é nossa

classe do componente. Vamos adaptar nosso código do template `cliente-add-edit.page.html` para o que apresento na sequência.

```
<ion-content padding-top>
  <form (ngSubmit) = "submit()">
    <ion-item lines="none">
      <ion-label position="floating">Nome</ion-label>
      <ion-input [(ngModel)]="nome" name="nome" type="text"></ion-input>
    </ion-item>
    <!-- demais itens -->
    <ion-button shape="round" color="primary" expand="block" padding
type="submit">
      Confirmar
    </ion-button>
  </form>
</ion-content>
```

Observe a definição de um elemento `<form>` com um *event binding* para `ngSubmit`, que invocará o método `submit()` quando o `<ion-button>` for pressionado. Veja, no botão, a definição de `type="submit"`, o que torna o componente o responsável pelo envio dos dados englobados pelo formulário para o método definido em `form`. Peço que veja também que no `<ion-input>` adicionamos agora um novo atributo, o `name`, que é uma exigência do Ionic/Angular quando temos componentes de entrada de dados dentro de um `<form>`. Isso possibilitará uma varredura, pelo framework, no DOM dos componentes filhos do formulário.

Vimos anteriormente que podemos criar propriedades em nossa classe que recebam e forneçam os dados de e para nossos componentes visuais. Seguindo o modelo de implementação que estamos trabalhando, teríamos uma propriedade para cada `<ion-input>`. Entretanto, se pensarmos em um contexto orientado a objetos, todos os dados solicitados pelo formulário pertecem a um conjunto comum, que são os dados de um cliente, o que nos leva a uma classe.

No momento, não criaremos classes para nosso modelo de negócio `Cliente`. Vamos fazer uso de um recurso do JavaScript, de criar um

objeto que terá suas propriedades baseadas nos dados vindos do template. Veja na sequência a declaração da propriedade `cliente` como um objeto JavaScript, vazio, que deve ser feita antes do construtor, em nossa classe.

```
cliente = {};
```

Na sequência, precisamos ajustar nosso template, que precisará fazer uso dessa nossa propriedade. Veja o código para `nome`, na sequência. Veja que o valor para o *data binding* agora é composto pelo nome de nossa propriedade e da propriedade que queremos utilizar para o nosso objeto `cliente`.

```
<ion-input [(ngModel)]="cliente.nome" name="nome" type="text"></ion-input>
```

Depois de alterar todos os `<ion-input>`, inserindo uma propriedade para `cliente`, de acordo ao solicitado no componente, altere o método `submit()` para o código a seguir.

```
submit() {
    console.log(this.cliente);
}
```

Informe os dados nos componentes, clique/pressione o botão de confirmação e veja o resultado na console do navegador, tal qual mostra a figura a seguir. Veja os nomes das propriedades para `cliente`.

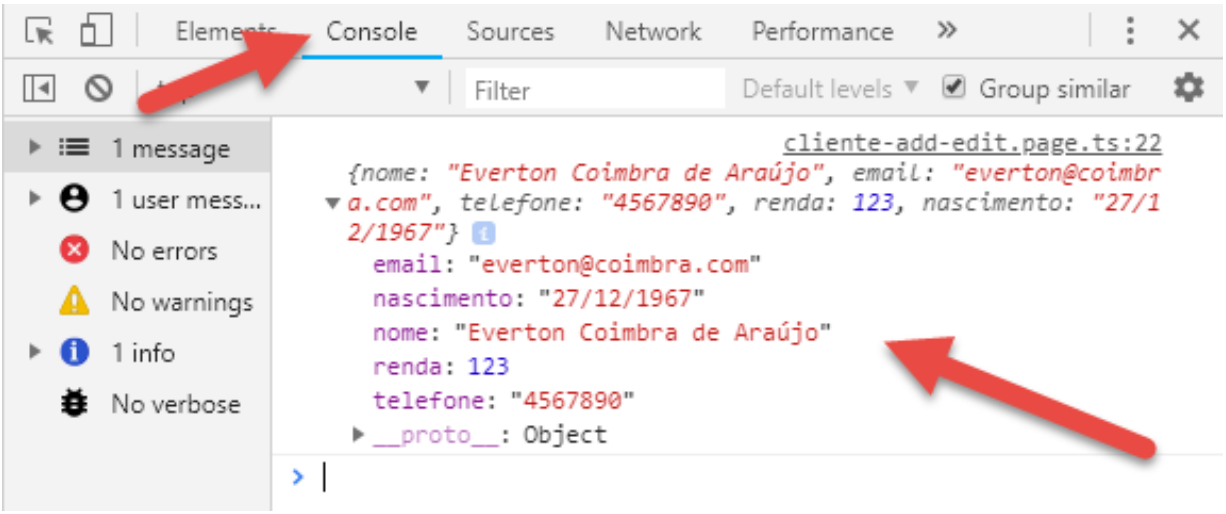


Figura 3.5: Estrutura criada para o projeto

A técnica que utilizamos na seção que estamos terminando agora é conhecida como `Template Driven Forms`, onde temos nosso formulário guiado por um modelo, nosso template. Vimos que nos controles indicamos o *data binding* por meio do `ngModel` e que recuperamos em nossa classe os dados informados no template, mas, que também podemos popular nossos componentes com valores atribuídos em nossa propriedade de classe.

3.8 Angular Reactive Forms

De acordo com o site do Angular, o Reactive Forms fornece uma estratégia guiada ao modelo para gerenciar entrada de dados em um formulário onde estes valores sofrem alteração, quer seja de maneira interna, ou por interação com o usuário. Vamos adaptar nosso componente de clientes para utilizar este recurso.

Nosso primeiro passo é compreender como podemos adicionar o Reactive Forms à nossa aplicação. É preciso entender que o Ionic/Angular trabalham com o conceito de módulos, e, nestes módulos, temos componentes. Quando criamos nossa página de

clientes usamos o Ionic CLI para gerar um componente, que é do tipo `page`. Veja a pasta do componente e verifique a existência de um arquivo chamado `cliente-add-edit-module.ts`. Com este arquivo aberto, observe a declaração da propridade `declarations` no objeto enviado ao decorator `@NgModule()`. Esta instrução informa ao Ionic que o módulo `ClienteAddEditPageModule`, que é uma classe, declara que tem implementado nele a classe `ClienteAddEditPage` que, se você observar, está declarada nos imports, trazendo para a referência dela o artefato `cliente-add-edit.page`.

Temos então um componente que é composto por uma classe TypeScript e um template HTML, que pertence a um módulo. Nossa situação agora é a seguinte: precisamos usar o `Angular Reactive Forms`, que é um componente, mas, para termos acesso a um componente, precisamos ter acesso ao módulo a que ele pertence. Desta maneira, voltando ao nosso módulo de clientes, observe a propriedade `imports` no decorator `@NgModule()`. Temos importados vários componentes. Precisamos informar que necessitamos importar para nosso módulo o módulo responsável pelo `Reactive Forms`. Sendo assim, adicione a seguinte instrução ao final do `imports`. Fique atento à importação que será necessária.

```
ReactiveFormsModule
```

Perfeito, já podemos trabalhar no uso do `Reactive Forms` em nosso componente. Abra o arquivo `cliente-add-edit-page.ts`. Precisamos declarar uma propriedade que gerenciará nosso formulário. Também precisaremos ter em nossa classe um objeto `FormBuilder` e receberemos este objeto pelo construtor. Veja o código a seguir.

```
clienteForm: FormGroup;

constructor(private formBuilder: FormBuilder) {
}
```

A primeira explicação que veremos é o porquê de `private` antes do argumento. A resposta é simples e está relacionada a escopos. Se não utilizarmos um modificador de escopo, o argumento terá

visibilidade apenas dentro do construtor. Ao utilizarmos `private`, estamos informando que o argumento será visto como um campo privado para o componente, ou seja, visível na classe. Poderíamos usar `public`, o que transformaria o argumento em propriedade, visível por classes consumidoras e também pelo nosso template.

Quando temos um formulário HTML, temos dentro dele o que chamamos de controles filhos, pertencentes ao formulário e que normalmente possuem os valores informados pelo usuário. Estes controles filhos, no `Reactive Forms`, são chamados de `FormControl`. Um objeto `FormGroup` agrega nele os valores de todos os `FormControls` pertencentes ao formulário em questão.

O `FormBuilder` é uma classe que possibilita a criação de diversos componentes relacionados ao `Reactive Forms`, dentre eles, o `FormGroup`. Normalmente, quando precisamos de algum serviço oferecido pelo framework, nós não vamos buscar por ele, apenas informamos que precisamos dele e que ele deve ser fornecido para o componente que informa esta necessidade. Isso é conhecido como injeção de dependência e essa injeção é realizada pelo construtor. Desta maneira, quando declaramos, no código anterior, que receberemos um argumento chamado `formBuilder`, ao tipificarmos, o framework já o fornecerá para nosso componente e o disporá como uma propriedade privada de nossa classe.

Com a declaração de nosso `FormGroup` e `FormBuilder` realizada, precisamos agora atribuir um objeto de grupo à nossa propriedade `clienteForm`. Se você conhece Orientação a Objetos, talvez tenha a tentação de realizar esta configuração no construtor, mas o padrão no Ionic/Angular (e também recomendado para outras linguagens) é realizarmos isso em um método de inicialização. Se você olhar atentamente para a declaração de nossa classe, verá que ela implementa a interface `OnInit` e que, quando geramos a página pelo Ionic CLI, um método chamado `ngOnInit()`, desta interface, está declarado, mas sem comportamento. É nele que configuraremos nosso formulário. Veja o código a seguir. Pode ser necessário importar `Validators`.

```
ngOnInit() {
    this.clienteForm = this.formBuilder.group({
      nome: ['', Validators.compose([Validators.required,
Validators.minLength(3), Validators.maxLength(50)])],
      email: ['', Validators.compose([Validators.required,
Validators.email])],
      telefone: ['', Validators.required],
      renda: ['0', Validators.compose([Validators.required,
Validators.min(0)])],
      nascimento: ['', Validators.required]
    });
}
```

Veja a invocação ao método `group()`, enviando a ele um objeto JavaScript, com propriedades nomeadas de acordo com o que teremos definido em nosso template e, ainda, adicionando validadores padrões para estas propriedades. Em algumas, temos apenas uma validação a ser realizada, em outras, temos mais de uma, e fazemos uso do método `compose()` para isso. Veja que, para `renda`, estamos declarando um valor inicial, `0`, para ser exibido no controle. O `Reactive Forms` traz um conjunto de validadores já implementados e à disposição, mas também nos dá a possibilidade de criação de validadores personalizados.

Como mudamos a maneira de ligação de nossos controles de interação com nosso componente, precisamos alterar nossos controles no template. Veja o código a seguir.

```
  <form [formGroup]="clienteForm" (submit)="submit()">
    <ion-item lines="none">
      <ion-label position="floating">Nome</ion-label>
      <ion-input formControlName="nome" type="text"></ion-input>
    </ion-item>
     <ion-item lines="none">
        <ion-label position="floating">Email</ion-label>
        <ion-input formControlName="email" type="email"></ion-input>
    </ion-item>
    <ion-item lines="none">
      <ion-label position="floating">Telefone</ion-label>
      <ion-input formControlName="telefone" type="tel"></ion-input>
```

```
    </ion-item>
    <ion-item lines="none">
      <ion-label position="floating">Renda</ion-label>
      <ion-input formControlName="renda" type="number"></ion-input>
    </ion-item>
    <ion-item lines="none">
      <ion-label position="floating">Nascimento</ion-label>
      <ion-input formControlName="nascimento" type="text"></ion-input>
    </ion-item>
    <ion-button shape="round" color="primary" expand="block" padding
type="submit">
      Confirmar
    </ion-button>
  </form>
```

Temos um *data binding* para `formGroup`, que recebe nossa variável definida no componente e, a mudança para o *event binding*, que agora é o `submit`. Esta é a sintaxe para um formulário utilizando o `Reactive Forms`. Outra mudança está nos `<ion-input>`, que não têm mais binding com um objeto do componente e tampouco a definição de `name`. No lugar disso, temos a definição de `formControlName`, recebendo o nome da propriedade enviada na criação de nosso `formGroup` para a variável `clienteForm`. Estes nomes devem ser IDÊNTICOS.

Nós temos toda a burocracia criada para binding do formulário com nosso componente e também temos as validações, mas, se clicarmos/pressionarmos o botão, nada ocorre. Você pode pensar: vamos aplicar as validações no momento em que os dados forem submetidos. E você não está errado, esta é uma possibilidade, sim. Mas vamos utilizar um recurso de validação "in-line" dos controles. Insira o código a seguir abaixo dos respectivos `<ion-input>`.

```
<!-- NOME -->
<small class="error-message"
    *ngIf="clienteForm.get('nome').errors?.required ||
clienteForm.get('nome').errors?.minlength ||
clienteForm.get('nome').errors?.maxlength">
    Informe o nome (3 a 50 caracteres)
```

```
</small>

<!-- EMAIL -->
<small class="error-message"
    *ngIf="clienteForm.get('email').errors?.required ||
clienteForm.get('email').errors?.email">
    Informe um email válido
</small>

<!-- TELEFONE -->
<small class="error-message"
    *ngIf="clienteForm.get('telefone').errors?.required">
    Informe o telefone
</small>

<!-- RENDA -->
<small class="error-message"
    *ngIf="clienteForm.get('renda').errors?.required ||
clienteForm.get('renda').errors?.min ">
    Informe uma renda positiva
</small>

<!-- NASCIMENTO -->
<small class="error-message"
    *ngIf="clienteForm.get('nascimento').errors?.required">
    Informe uma data de nascimento
</small>
```

Note que ao utilizarmos a tag HTML `<small>`, estamos utilizando uma classe de estilo. Precisamos criá-la em nosso arquivo de CSS, tal qual podemos ver na sequência.

```
small.error-message {
    color: var(--ion-color-danger);
}
```

Ainda, em relação à tag `<small>`, do código anterior, observe que utilizamos uma diretiva do Angular, chamada `*ngIf`. Esta diretiva recebe o retorno de uma expressão lógica e, se este retorno for verdadeiro, o componente que a declara é renderizado. Você pode

verificar que nossa expressão está relacionada à identificação de existência de erros relacionados às validações registradas para cada propriedade do formulário. Se o valor for verdadeiro, então, como dito, o texto de alerta é exibido. Veja a figura a seguir, que apresenta nosso template em execução e com as validações aplicadas.


12:34 PM
100%
Registro de CLIENTES
Nome
Informe o nome (3 a 50 caracteres)
Email
Informe um email válido)
Telefone
Informe o endereço
Renda
-1
Informe uma renda positiva
Nascimento
Informe uma data de nascimento
Confirmar

Figura 3.6: Validação para os controles de entrada do template

Para finalizar a explicação desta técnica de validação, podemos implementar um código que habilite o botão de confirmação dos dados apenas quando o formulário não possuir mais erros. Para isso, veja o código para o botão na sequência. Identifique nele o binding com `disabled` e a validação do formulário.

```
<ion-button shape="round" color="primary" expand="block" padding
type="submit" [disabled]="!clienteForm.valid">
      Confirmar
</ion-button>
```

3.9 Não exibir mensagens de erro antes da submissão e alteração

Com a técnica que trabalhamos na seção anterior, assim que acessamos nossa interface com o usuário, já nos deparamos com as mensagens de erro, pois, a princípio, os controles não estão preenchidos e a mensagem é então exibida. Mas e se quisermos que essa mensagem seja exibida apenas após termos tentado submeter os dados informados?

Para implementarmos a exibição de mensagens de validação apenas após a tentativa de submissão precisamos, inicialmente, tirar do botão de confirmação a validação que temos para habilitação/desabilitação dele, pois o usuário precisará submeter os dados para que eles possam ser verificados e, em caso de requisitos não cumpridos, as mensagens sejam então exibidas. Desta maneira, vamos deixar nosso componente de botão tal qual o código a seguir.

```
<ion-button shape="round" color="primary" expand="block" padding
type="submit">
      Confirmar
</ion-button>
```

Na sequência, precisamos colocar a validação retirada do botão no método que captura a submissão de nossos dados. Veja essa implementação na sequência. Verifique nela que, caso haja algum problema com os dados do formulário da página, a execução do método é interrompida. Poderíamos, desta maneira, implementar o processamento dos dados após o `if()`, o que garante que ele ocorrerá apenas quando os dados forem válidos.

```
async submit() {
    if (this.clienteForm.invalid || this.clienteForm.pending) {
      return;
    }
}
```

Com a implementação que fizemos, o problema ainda não está resolvido, pois as mensagens de erro continuam sendo exibidas logo que a página é exibida. Precisamos de uma variável de controle, que identificará quando nosso formulário já teve uma submissão realizada. Vamos então fazer uso de uma variável de template em nosso componente de formulário. Veja o código a seguir.

```
<form #form="ngForm" [formGroup]="clienteForm" (submit)="submit()">
```

Observe a declaração `#form="ngForm"`. Com esta instrução, estamos atribuindo a `form` o objeto que representa o formulário, via Angular, por meio do `ngForm`. Com isso, temos acesso a informações do formulário e poderemos utilizá-las para a exibição ou não das mensagens de erro apenas após a tentativa de submissão. Veja na sequência a adaptação para o `*ngIf` do componente de nome.

```
<small class="error-message"
        *ngIf="(clienteForm.get('nome').errors?.required ||
clienteForm.get('nome').errors?.minlength ||
clienteForm.get('nome').errors?.maxlength) &&  form.submitted">
    Informe o nome (3 a 50 caracteres)
</small>
```

Englobamos as condicionais de requisitos de validação entre parênteses e adicionamos um condicional `&&` para `form.submitted`. A mensagem só será exibida se os dados estiverem inválidos e se o formulário já tiver sido submetido. O que acha de adaptar os demais controles e testar a aplicação?

Seguindo esta linha de exibição de mensagem apenas após algum tipo de interação, como fizemos para a submissão, podemos ter a possibilidade de exibição da mensagem após o usuário sair do controle visual. Isso é também possível e bem simples. Agora, para variarmos, vamos ver esta alteração no componente para telefone. Veja o código a seguir.

```
<small class="error-message"
      *ngIf="clienteForm.get('telefone').errors?.required &&
(form.submitted || clienteForm.get('telefone').touched)">
    Informe o telefone
</small>
```

Verifique que inserimos parênteses para o `form.submitted` e adicionamos um `OU` para verificar se a propriedade do formulário, chamada `telefone`, foi `tocada`, ou seja, passou por um processo de validação. Vamos testar nos demais controles?

Vamos aproveitar e ver um novo recurso visual para apresentar controles com regras não cumpridas em relação às validações. Veja a seguir a nova escrita para o `<ion-input>`. Observe o binding para uma classe CSS chamada `invalid` e note as regras que farão com que ela seja utilizada.

```
<ion-input formControlName="nome" type="text"
[class.invalid]="!clienteForm.controls['nome'].valid &&
clienteForm.controls['nome'].touched"></ion-input>
```

Precisamos, para utilizar a classe de estilo `invalid`, implementá-la no arquivo de estilos de nossa página. Veja a seguir o código para ela, que podemos inserir ao final de nosso arquivo. Apenas mais uma dica de validação e integração de nossa página com CSS.

```
.invalid {
    border-bottom: 1px solid #ff6153;
}
```

3.10 Validação do Reactive Form no componente TypeScript

Vimos na seção anterior que uma das possibilidades de validação de um formulário é exibir, próximo a seu componente de entrada, uma mensagem de aviso ao usuário sobre as validações não cumpridas. Durante os testes, você pode ver que conforme atendem às validações, as mensagens desaparecem, pois estão condicionadas à existência do erro. Trabalhamos também a situação de habilitar o botão de confirmação apenas para quando todas as validações tiverem sido atendidas.

Mas, e se a necessidade, ou desejo, for de exibir os erros apenas após o usuário clicar/pressionar o botão de confirmação? Bem, se você notou a condição de não habilitação do botão, ela faz uso de uma propriedade do formulário que nos informa esta situação e podemos, inicialmente, trabalhar com ela.

Para testarmos isso, vamos comentar nossos elementos `<small>` e, em nosso componente, vamos criar uma propriedade, que indicará a existência de erros durante o processo de validação e outra para receber a mensagem de erro. A propriedade `hasErrors` será utilizada como condição para exibirmos, ou não, mensagens de erro de validação. Precisamos também deixar habilitado nosso `<ion-button>`, desta maneira, retire o *data binding* que controla o estado dele, de acordo às validações do formulário. Veja a implementação na sequência, que é seguida de uma implementação a ser realizada em nosso template, logo após a definição de nosso `<form>`.

```
// arquivo TS, antes do construtor
hasErrors = false;
```

```
errorMessage: string;

<!-- arquivo HTML, após <form> -->
<ion-label *ngIf="hasErrors">
  Erro
    {{ errorMessage }}
</ion-label>
```

Notou que inicializamos a propriedade `hasErrors` com `false`? Isso impedirá, de imediato, que o `<ion-label>` de erros seja exibido. Já dentro do componente, temos uma `Angular Expression`, que contém a propriedade a ser invocada e ter seu resultado renderizado. A sintaxe é colocar a expressão entre chaves duplas, `{{}}`.

Vamos agora implementar uma técnica de validação no método que será invocado na submissão do formulário, o nosso já conhecido `submit()`. Veja o código para ele na sequência.

```
submit() {
    if (this.clienteForm.invalid) {
      this.hasErrors = true;
      this.errorMessage = 'Existem erros nos dados informados';
    } else {
      this.hasErrors = false;
    }
}
```

Verificamos se o formulário `clientesForm` é inválido e, caso seja, atribuímos `true` a `hasErrors` e uma mensagem a `errorMessage`. Caso não existam erros no formulário, por segurança, a variável de controle recebe `false`. Após esta validação, poderemos delegar a continuidade do fluxo do processo. Teste sua aplicação e, sem informar nada, clique/pressione o botão de confirmação.

Mas não tem como realizar as validações para cada uma das que configuramos para o formulário? É claro que tem! Vamos a algumas alterações para isso. A primeira, será alterar o tipo de dado de nossa `errorMessage`, para uma matriz de `string`. Veja o código a

seguir e observe que aproveitei para pluralizar o nome da propriedade.

```
errorsMessage: string[];
```

Com esta alteração, vislumbramos um elemento da matriz para cada erro que possa existir durante o processo de validação. Vamos então à nova implementação para o método `submit()`. Veja-a na sequência.

```
submit() {
    this.errorsMessage = [];
    if (this.clienteForm.get('nome').hasError('required')) {
      this.errorsMessage.push('Nome é obrigatório');
    }
    if (this.clienteForm.get('email').hasError('required')) {
      this.errorsMessage.push('Email é obrigatório');
    }
    this.hasErrors = this.errorsMessage.length > 0;
}
```

Observe, após a inicialização da propriedade `errorsMessage`, uma condição que verifica a existência de um erro para a validação `required` para uma propriedade de `clienteForm`, chamada `nome`. Em caso positivo, adicionamos à matriz de mensagens de erro uma string apontando isso. Fazemos o mesmo para o `email`. Ao final, temos a propriedade `hasErrors` recebendo um valor booleano, de acordo com a existência ou não de elementos na matriz de erros. Se um erro for adicionado, ele(s) deve(m) ser exibidos para o usuário.

Mas como exibir agora os vários possíveis erros? Poderíamos apenas ajustar o nome na Angular Expression para a nova propriedade. Entretanto, as mensagens apareceriam uma do lado da outra, e nosso objetivo é que elas apareçam uma embaixo da outra.

O que precisamos é de uma maneira de exibir o conteúdo da propriedade como uma listagem, linha a linha. O Angular/Ionic nos oferece uma diretiva, `*NgFor`, que nos possibilita uma iteração em

uma coleção de dados, isto é, poderíamos ter um `<ion-label>` para cada elemento da propriedade. Nosso código poderia ser o que está na sequência, em substituição ao que já temos para a exibição do erro.

```
<ion-label *ngFor="let errorMessage of errorsMessage">
  {{errorMessage}}<br/>
</ion-label>
```

Esta diretiva executa uma varredura, elemento a elemento de uma coleção, em nosso caso, `errorsMessage` e, para cada elemento, a cada iteração, ocorre seu armazenamento, por meio do `let` , em uma variável chamada de `errorMessage` . Desta maneira, podemos ter a Angular Expression apontada no código anterior, que será exibida quantas vezes for a quantidade de elementos na coleção. O que acha de testar a aplicação?

Você deve ter percebido pelo código que, ao termos a página renderizada, antes de confirmarmos os dados, um `<ion-label>` foi renderizado. Isso é algo esperado, pois não temos uma condição para exibição ou não dos erros. Se você confirmar, os erros serão exibidos, um em cada linha. Mas não queremos perder o espaço ocupado por elemento inexistente quando a página for renderizada. O que podemos fazer?

Para evitarmos o problema do parágrafo anterior, você possa talvez ter pensado na `*NgIf` , que já utilizamos. Você está correto, mas não é possível termos duas diretivas em um mesmo componente Ionic. Como queremos uma listagem, onde cada `<ion-label>` esteja em uma linha, podemos utilizar o componente `<ion-list>` e nele realizar a verificação de exibição ou não e, dentro dele, ter o `<ion-label>` . Veja nosso código na sequência. Teste sua aplicação após a implementação e veja o resultado antes e após a confirmação dos dados.

```
<ion-list *ngIf="hasErrors">
    <ion-label *ngFor="let errorMessage of errorsMessage">
      {{errorMessage}}<br/>
```

```
    </ion-label>
</ion-list>
```

O que acha de estilizarmos essa exibição de erros? Na sequência, uma sugestão, apenas para você verificar a possibilidade. Lembre-se de que este código deve estar em nosso arquivo de estilos.

```
ion-list.errors-summary {
    border: 1px solid black;
    background-color: red;
    margin-bottom: 0px;
}

ion-label.errors-summary {
    color: white;
    text-align: center;
    margin-bottom: 0px;
    margin-top: 0px;
}
```

Após a implementação dos estilos, precisamos utilizá-los, sendo assim, insira o uso das classes nos componentes com a instrução `class="errors-summary"` dentro das tags iniciais deles. Teste novamente sua aplicação e veja o resultado.

Podemos aplicar outra técnica para exibição dos erros de validação, sem a necessidade de fazermos verificações na submissão do formulário. Ela pode ser um pouco mais burocrática, mas pode ser interessante de conhecermos. Comente em seu código, tanto do template HTML, como no componente TypeScript, nossos tratamentos de erro, para testarmos este novo.

Para faciliar, na classe comentei `hasErrors = false;` e `errorsMessage: string[];`. Consequentemente, comentei o código do método `submit()` e o `<ion-list>` da mensagem de erro após o `<form>`.

Vamos começar definindo uma propriedade de um objeto JavaScript. Este objeto terá propriedades de acordo ao nome das propriedades de nosso formulário. Cada propriedade dessa terá

como valor uma matriz de objetos, composta pelo tipo de validação que cada componente do formulário tem, e de uma mensagem de erro a ser aplicada quando a validação for malsucedida. Veja o código na sequência. Implementei apenas para `nome` e `renda`. Você pode, se quiser, implementar para as outras propriedades do formulário, que deve estar antes do construtor, junto com as demais propriedades.

```
validationMessages = {
    nome: [
        { type: 'required', message: 'Nome é obrigatório' },
        { type: 'minlength', message: 'Nome deve ter ao menos 3
caracteres' },
        { type: 'maxlength', message: 'Nome não pode ter mais que 50
caracteres' }
      ],
    renda: [
        { type: 'min', message: 'Renda precisa ser positiva' }
      ]
};
```

Agora que temos as validações mapeadas para um objeto específico de mensagens de erro, precisamos ajustar nosso template para fazer uso deste objeto e de nossas validações. Sendo assim, insira, logo após de `<form>`, o código a seguir.

```
<div>
  <ng-container *ngFor="let validation of validationMessages.nome" >
    <div *ngIf="clienteForm.get('nome').hasError(validation.type)">
      {{ validation.message }}
    </div>
  </ng-container>
  <ng-container *ngFor="let validation of validationMessages.renda" >
      <div *ngIf="clienteForm.get('renda').hasError(validation.type)">
        {{ validation.message }}
      </div>
  </ng-container>
</div>
```

Consegue abstrair o código anterior? No exemplo, deixei de utilizar componentes Ionic, pois essa é uma possibilidade que podemos ter também, utilizarmos HTML e componentes Angular. Usamos as diretivas `*NgFor` e `*NgIf`, que já conhecemos. Execute sua aplicação e teste. Veja que os erros que existem, já aparecem no topo da página e, conforme formos informando os valores corretos, os erros deixam de ser exibidos.

3.11 Um componente específico para data

Como você certamente realizou diversos testes em nossa aplicação, já notou que a informação da data de nascimento do cliente ocorre em um controle de entrada normal para texto. Teríamos que informar a data em um formato e valor corretos. Mas, usando algumas aplicações em nossos dispositivos, vemos a existência de um controle especial para informação de uma data. O Ionic nos oferece isso também. Substitua o `<ion-input>` de nascimento pela instrução a seguir. Observe os atributos para `displayFormat`, `min` e `max`.

```
<ion-datetime formControlName="nascimento" displayFormat="DD/MM/YYYY"
min="1967" max="2020-10-31"></ion-datetime>
```

Vamos agora executar nossa aplicação. Observe que o componente é renderizado da mesma maneira no Android e iOS. Mudam alguns detalhes visuais, que são padrões para cada plataforma, mas não é feito uso dos recursos nativos delas, não apresentam o visual específico para este controle. Precisamos trazer para nossa aplicação componentes que possibilitem este acesso nativo.

Felizmente, podemos fazer uso de plugins, e existem vários, que você pode verificar em https://ionicframework.com/docs/native/. Nós faremos uso agora do `Date Picker`, que é um plugin Cordova. No terminal, na pasta de seu projeto, execute as instruções a seguir.

```
// Integração para o Cordova, caso não tenha feito ainda
ionic integrations enable cordova --add

// Instalação do plugin
ionic cordova plugin add cordova-plugin-datepicker

// Instalação do plugin no projeto. Observe a versão que usaremos
npm install @ionic-native/date-picker@5.7.0

// Caso não tenha instalado este plugin ainda.
npm i @ionic-native/core@5.7.0
```

Após a instalação ter ocorrido com sucesso, precisamos informar à nossa aplicação que faremos uso deste plugin. Um plugin também é visto como um módulo para as aplicações Ionic. Um detalhe importante para o caso do `DatePicker` é que ele será visto como um serviço/recurso, que poderá ser injetado em outros componentes. Desta maneira, precisamos registrá-lo no arquivo `app.module.ts`, na propriedade `providers`, mas, antes disso, precisamos importá-lo para nosso módulo principal, o `app.module.ts`. Veja a instrução a seguir, a ser inserida no início do arquivo.

```
import { DatePicker } from '@ionic-native/date-picker/ngx';
```

Agora sim, vamos então inserir `DatePicker` após `SplashScreen` no elemento `providers`. Para auxiliar, na sequência, veja este elemento com a inserção.

```
providers: [
    StatusBar,
    SplashScreen,
    DatePicker,
    { provide: RouteReuseStrategy, useClass: IonicRouteStrategy }
],
```

Precisamos agora realizar a implementação deste plugin em nosso componente/página de clientes. Não temos um componente Ionic (tag) para inserirmos este plugin em nosso template, o que nos levará a utilizar o atual componente que temos, o `<ion-datetime>`, que receberá a data que selecionarmos em nosso `DatePicker`. Veja

que o componente apenas será utilizado para exibir a data selecionada, não poderemos permitir que o usuário interaja com ele. Sendo assim, vamos inserir o seguinte atributo no componente, que deixará o controle desabilitado.

```
<ion-datetime formControlName="nascimento" displayFormat="DD/MM/YYYY"
min="1967"
max="2020-10-31" [disabled]="true">
</ion-datetime>
```

Na sequência, será necessário estabelecer um meio para a invocação de um método de nosso componente, que será responsável por renderizar o `DatePicker`. Inicialmente, faremos isso no `<ion-item>` do `<ion-datetime>`. Veja o código na sequência.

```
<ion-item lines="none" (click)="selecionarData()">
```

Agora, resta-nos implementar este método em nossa classe responsável pelo template. Porém, antes disso, precisamos ter injetado em nossa classe o `DatePicker` e, para isso, precisamos inserir a seguinte declaração em nosso construtor. Veja o código na sequência.

```
private datePicker: DatePicker
```

Com o objeto/serviço/componente disponibilizado para nossa classe, vamos então implementar o método `selecionarData()`:

```
selecionarData() {
    this.datePicker.show({
      date: new Date(),
      mode: 'date',
      locale: 'pt-br',
      doneButtonLabel: 'Confirmar',
      cancelButtonLabel: 'Cancelar'
    })
    .then(
      data =>
this.clienteForm.controls.nascimento.setValue(data.toISOString())
```

```
    );
}
```

Verifique que invocamos o método `show()`, enviando a ele um objeto JavaScript com configurações que personalizarão a renderização do componente. Este método retorna um `Promise<Date>`, onde `Date` é utilizado como tipo genérico para `Promise`. Mas o que é uma `Promise`?

Quando executamos uma tarefa de maneira síncrona, aguardamos por ela terminar seu processo antes da execução do código passar para uma seguinte instrução. Quando precisamos que as tarefas sejam executadas, sem dar a impressão de congelamento da aplicação, precisamos fazer uso de programação e execução assíncrona.

O ECMAScript 6 (ES6) trouxe uma API conhecida como `Promise`, que possibilita a resolução destas chamadas. Podemos então dizer que uma `Promise` é um retorno à chamada a uma função assíncrona que, ao ser concluída, devolverá ao chamador, o fluxo da execução da aplicação e então, com o retorno à chamada, por meio de funções de callback, executamos funcionalidades que deveriam ser executadas quando a `Promise` for concluída.

A arrow function que temos em `then()` será executada apenas quando o método `show()` for totalmente concluído. Em nosso caso, é a renderização do componente para seleção de uma data. Veja que `then()` recebe um valor para a variável `data`, que será a data selecionada. Esta data então é formatada por `toISOString()` e atribuída para nosso componente de formulário relacionado à data de nascimento, que é nosso `<ion-datetime>`, que renderizará a data escolhida.

Retornemos ao objeto enviado para configuração. A propriedade `date` receberá a data atual como data inicial a ser exibida; `mode` recebe o que deverá ser renderizado pelo controle, em nosso caso, datas. `locale` trabalha a localização a ser utilizada pelo controle, em

nosso caso, Português/Brasil; enfim, `doneButtonLabel` e `cancelButtonLabel` são semanticamente entendidos.

Precisamos realizar algumas alterações no template. Troque o `position` do `<ion-label>` para `stacked`, pois não teremos o `<ion-datetime>` com interação, então melhor deixarmos o label já acima da data. O que acha agora de executarmos a aplicação para testarmos?

Verificando o console do navegador, na execução da aplicação, podemos notar uma mensagem de alerta, tal qual se segue.

```
It looks like you're using the disabled attribute with a reactive form directive. If you set disabled to true when you set up this control in your component class, the disabled attribute will actually be set in the DOM for you. We recommend using this approach to avoid 'changed after checked' errors.
```

A mensagem anterior é uma recomendação, para tirarmos o `[disabled]="true"` de nosso `<ion-datetime>` e inserirmos na definição do controle nos objetos de nosso formulário para o Reactive Forms. Veja na sequência a alteração para `nascimento`.

```
nascimento: [{value: '', disabled: true}, Validators.required]
```

Se formos executar a aplicação no navegador, teremos erro de execução ao clicarmos em qualquer parte do `<ion-item>`, pois o componente é nativo e precisa estar executando em um emulador ou dispositivo. Este erro pode ser visto no console do navegador.

Mas e se quisermos ter, para os dispositivos, o controle nativo e, para o navegador, o controle Ionic? Isso é possível e é tranquilo. Precisaremos trabalhar com a identificação da plataforma onde a aplicação está sendo executada e veremos isso agora.

Nosso primeiro passo é injetar em nosso componente um serviço que nos possibilitará a identificação da plataforma que o aplicativo está sendo executado. Insira no construtor, entre os argumentos, a

instrução a seguir. Será preciso adicionar `Platform` ao import do `@ionic/angular`.

```
private platform: Platform
```

Agora, para evitarmos o erro em tempo de execução que temos, precisamos, antes de invocar o método do `DatePicker`, verificar a plataforma e, se for a desejada, executamos o código que trabalhamos há pouco para renderização do controle. Veja o novo código para nosso método `selecionarData()` na sequência.

```
selecionarData() {
    this.platform.ready().then(() => {
      if (this.platform.is('cordova')) {
        this.datePicker.show({
          date: new Date(),
          mode: 'date',
          locale: 'pt-br',
          doneButtonLabel: 'Confirmar',
          cancelButtonLabel: 'Cancelar'
        })
        .then(
          data =>
this.clienteForm.controls.nascimento.setValue(data.toISOString())
        );
      } else {
        // instruções para execução no navegador
      }
    });
}
```

Veja que agora o início do método é a execução de uma `Promise` que, ao ser concluída, verificamos a plataforma em execução e então inserimos nosso já conhecido código. Deixei o `else` para sabermos dessa possibilidade. Se quiser verificar as plataformas possíveis de serem identificadas, acesse https://ionicframework.com/docs/api/platform/Platform/. Tente executar sua aplicação no navegador agora. Clique sobre o `<ion-item>`. Veja que não temos mais o erro de execução no console.

Ainda temos nosso `<ion-datetime>` desabilitado mesmo executando no navegador. Precisamos agora condicionar este estado do componente à plataforma em execução. Em nosso componente, vamos implementar a seguinte propriedade de leitura.

```
get isBrowserPlatform(): boolean {
    if (this.platform.is('cordova')) {
      return false;
    }
    return true;
}
```

Trabalhamos, em situações diferentes, a diretiva `*NgIf` e, para efeito didático em um novo conhecimento, vamos usá-la agora, para resolver esta nossa situação, mas vamos utilizá-la com um `else`. Nosso objetivo é que, caso estejamos com a aplicação em uso em um dispositivo, o componente nativo seja renderizado, caso estejamos no navegador, o `<ion-datetime>` seja o componente renderizado. Veja então, na sequência, esta implementação.

```
<div *ngIf="isBrowserPlatform; else showNative">
  <ion-item lines="none">
    <ion-label position="floating">Nascimento</ion-label>
    <ion-datetime formControlName="nascimento" displayFormat="DD/MM/YYYY"
min="1967" max="2020-10-31"></ion-datetime>
    <small class="error-message"
      *ngIf="clienteForm.get('nascimento').errors?.required">
      Informe uma data de nascimento
    </small>
  </ion-item>
</div>

<ng-template #showNative>
  <div (click)="selecionarData()">
    <ion-item lines="none" padding-left>
      <ion-label>Nascimento</ion-label>
    </ion-item>
    <ion-item lines="none" padding-left>
      <ion-datetime disabled="true" placeHolder="Selecione a data"
formControlName="nascimento" displayFormat="DD/MM/YYYY" min="1967"
```

```
max="2020-10-31"></ion-datetime>
    </ion-item>
    <div text-center>
      <small class="error-message"
        *ngIf="clienteForm.get('nascimento').errors?.required">
        Informe uma data de nascimento
      </small>
    </div>
  </div>
</ng-template>
```

No início do código anterior, temos um `<div>` comum do HTML, mas que será renderizado apenas caso o método `get isBrowsePlatform` retornar verdadeiro, caso contrário, um template chamado `showNative` será renderizado. Este template está logo após o primeiro `<div>` .

Verifique o uso de `<ng-template>` e a maneira de declararmos um identificador para ele (`#showNative`). Temos nele um novo `<div>` com os componentes necessários para esta situação. Note que o método `selecionarData()` está ligado ao `click` do `<div>` .

Precisamos alterar o código que configura o controle `nascimento` no `Reactive Forms` , pois a desabilitação de nosso controle agora depende de verificação da plataforma. Poderíamos colocar o `disabled` no HTML, mas vimos que não é o recomendado. Veja a adaptação na sequência e teste sua aplicação.

```
nascimento: [{value: '', disabled: !this.isBrowserPlatform },
Validators.required]
```

3.12 Confirmando a inserção com o usuário

É sempre interessante uma boa interação com o usuário de nossos aplicativos e, informarmos para ele o sucesso, ou não, de uma operação, é extremamente relevante. Desta maneira, fechando este

capítulo, trabalharemos alguns recursos do Ionic para produzir informações de resultados de operações para o usuário.

O componente `Alert` é responsável pela renderização de um tipo de janela no dispositivo, com título, subtítulo, mensagem e botões. Existe uma série de recursos para o componente que podem ser utilizados e, aqui, trabalharemos algumas destas técnicas.

Precisamos injetar o `Alert` no construtor de nosso componente para então o utilizarmos. Será preciso importar o `AlertController`. Para facilitar, no código as seguir também está o `import` necessário.

```
import { AlertController } from '@ionic/angular';

private alertCtrl: AlertController
```

Com o `Alert` disponibilizado para nosso componente, criaremos um método responsável pela geração da mensagem de alerta ao usuário para então o consumirmos::

```
async presentAlert(header: string, subHeader: string, message: string,
buttons: string[]) {
    const alert = await this.alertCtrl.create({
      header,
      subHeader,
      message,
      buttons
    });
    await alert.present();
}
```

Observe a assinatura de nosso método. Veja o uso da instrução `async`, que torna nosso método assíncrono. Isso possibilita a continuidade de execução da aplicação, enquanto o método não é concluído, evitando o sentimento de "travamento" dela.

Os argumentos recebidos pelo método são os necessários para a criação de uma mensagem básica e seus nomes são semânticos, o que facilita nossa compreensão. Observe que a criação e apresentação da mensagem são precedidas da instrução `await`,

que andam em conjunto com o `async`. Sempre que temos chamadas `await` em um método, o método que engloba a instrução precisa ser declarado como `async`. Note que, como os argumentos recebidos têm o mesmo nome das propriedades do objeto enviado para `create()`, basta referenciá-los, sem precisar de uma atribuição.

Resumindo o método: recebemos os argumentos, os enviamos ao método `create()` de nosso objeto `alertCtrl` e, com objeto devidamente criado, invocamos a apresentação pela chamada a `present()`, de nosso objeto `alert`. Note que declaramos `alert` como `const`, pois não mudaremos o valor dela.

Resta-nos invocar nosso novo método e faremos isso na submissão de nosso formulário, no `submit()`. Veja a implementação e observe que precisamos alterá-lo para `async` e que a chamada ao método `presentAlert()` tem o `await`.

```
async submit() {
    this.errorsMessage = [];
    if (this.clienteForm.get('nome').hasError('required')) {
      this.errorsMessage.push('Nome é obrigatório');
    }
    if (this.clienteForm.get('email').hasError('required')) {
      this.errorsMessage.push('Email é obrigatório');
    }
    this.hasErrors = this.errorsMessage.length > 0;

    if (!this.hasErrors) {
      await this.presentAlert('Sucesso', 'Gravação bem sucedida', 'Os
dados do cliente foram gravados', ['Ok']);
    }
}
```

Vamos testar nossa aplicação. Preencha os dados do formulário e pressione o botão. Sua janela deve estar semelhante à apresentada na sequência.


12:34 PM
100%
Registro de CLIENTES
Nome
Ambrósio
Email
ambrosio@cc.com.br
Telefone
4512
Rend
23
Nasci
01/0
Sucesso
Gravação bem sucedida
Os dados do cliente foram gravados
OK
CONFIRMAR

Figura 3.7: Exibindo mensagens por meio do Alert

O Ionic traz outro recurso para interação com o usuário no que diz respeito à exibição de mensagens. É muito comum vermos em nossos dispositivos mensagens sendo exibidas, principalmente no topo da tela, conhecidas como notificações. Estas notificações são conhecidas também como `Toast`.

Assim como fizemos para o `Alert`, precisamos injetar em nosso componente o `ToastController`. Sendo assim, insira em nosso construtor, ao final, a instrução a seguir. Ajuste a importação.

```
private toastCtrl: ToastController
```

Na sequência, vamos implementar um método que será responsável pela apresentação de nossa mensagem ao usuário. Veja o código para ele na sequência.

```
async presentToast(message: string, duration: number, position: 'top' |
'bottom') {
    const toast = await this.toastCtrl.create({
      message,
      duration,
      position
    });
    toast.present();
}
```

A assinatura do método traz alguns argumentos básicos para a exibição do controle, que são: mensagem, duração em milissegundos, e a posição para exibição mensagem. Observe que o método também é assíncrono e a criação do componente é precedida pelo `await`. Ainda na assinatura, veja que limitamos os valores possíveis de serem recebidos para a posição do controle para `top` e `bottom`.

Para testarmos nossa nova implementação, no `submit()`, vamos trocar a invocação ao `Alert` para o `Toast`. Veja esta implementação na sequência, que finaliza então nossa seção e capítulo.

```
async submit() {
    await this.presentToast('Gravação bem sucedida', 3000, 'top');
}
```

Conclusão

Terminamos o capítulo com nossa primeira implementação de uma funcionalidade! Já vimos e trabalhamos bastante coisa. Conhecemos a estrutura básica de uma aplicação Ionic e definições sobre como os arquivos (artefatos) são gerados e ligados para o padrão de componentes trabalhado pelo framework. Pudemos interagir com o Ionic CLI para a criação de nossa página, que é vista como um componente em aplicações Ionic. Na página que criamos inserimos componentes de entrada e saída de dados, vimos possibilidades de uso de teclados diferentes para cada tipo de dado a ser informado. Configuramos alguns controles diretamente com diretivas Ionic e fazendo uso de classes de estilo (CSS).

Para iniciar nossa aplicação com a página que criamos, tivemos uma pequena introdução ao roteamento de páginas utilizado pelo framework. Descobrimos com agrupar controles por meio do componente `<ion-item>` e implementamos a classe do componente para acessar os valores informados nos controles de entrada. O Ionic oferece recursos para ligação de dados e eventos (data binding e event biding) e pudemos colocar isso em prática também neste capítulo. Finalizando o processo de entrada de dados, trabalhamos o `Angular Reactive Forms` , que traz, dentre seus recursos, possibilidades para validações sobre os dados informados e emissão de alertas ao usuário para o caso de erro na entrada. Dentre os tipos de dados que trabalhamos, tínhamos a data de nascimento, o que nos levou a conhecer um componente nativo para este tipo de dado, promovendo assim a experiência do usuário em seu dispositivo e recursos visuais oferecidos por sua plataforma. Finalizando o capítulo, trabalhamos dois componentes que exibem mensagens ao usuário, com vistas a informar o sucesso no processo de gravação. Foi um capítulo introdutório com muitos recursos. Procure executar sua aplicação agora com o Capacitor,

caso queira. Podemos agora dar uma relaxada, tomar uma água e então voltarmos para o próximo capítulo, onde trabalharemos a exibição de valores em forma de listagem.

Capítulo 4
Listagem de dados e navegação entre páginas

No último capítulo vimos vários recursos para a entrada de dados por parte do usuário. Não chegamos a persistir os dados informados, pois veremos isso no próximo capítulo, mas agora podemos verificar as possibilidades de exibição de dados armazenados localmente, em uma matriz e, depois, uniremos todos estes recursos atrelados à persistência e recuperação de dados.

Vou optar, para esta situação, pela criação de um novo projeto, até para podermos praticar estas funcionalidades. Entretanto, caso você queira, você pode dar continuidade ao projeto anterior.

A funcionalidade que implementaremos aqui está ligada ao registro dos tipos de serviços que a oficina pode prestar, consulta e alteração de tipos de serviços já registrados. Introduziremos também os componentes de serviço (services), oferecidos pelo Ionic/Angular.

4.1 Criação e execução inicial do projeto

Em sua pasta de projetos Ionic, insira no terminal, como sugestão, a seguinte instrução. Veja que nosso projeto agora se chama `capitulo04`.

```
// Lembre-se de estar na pasta raiz de seus projetos
ionic start capitulo04 blank --type=angular --no-git

// Acesse a pasta criada para seu projeto
npm install --save @ionic-native/core@5.7.0
```

Quanto à instalação da integração com o Cordova ou Capacitor, para execução em dispositivos ou emuladores, fica a seu critério, mas apontarei neste capítulo algumas situações relativas ao Capacitor, que está sendo desenvolvido como plataforma de integração nativa pelo Ionic, em substituição ao Cordova.

Estando no Visual Studio Code, com a pasta do projeto `capitulo04` aberta, vamos criar uma pasta chamada `pages`, dentro de `app`, no mesmo nível de `home` e, nesta pasta, uma chamada `tipo-servicos`. Depois, dentro desta nova pasta, vamos criar outra, chamada `listagem`. É nela que teremos os artefatos para implementação de nossa listagem para os tipos de serviços oferecidos pela oficina. Note que diferentemente do capítulo anterior, onde criamos as páginas diretamente em `app`, optamos agora por uma pasta específica para `pages`. Poderíamos ter seguido a mesma estratégia que tivemos antes, ao utilizar o Ionic CLI para criação de nossa página, mas quis criá-la agora diretamente no IDE, para que possamos conhecer esta possibilidade também.

Vamos criar três arquivos dentro desta nova pasta: `tipo-servicos-listagem.page.html`, `tipo-servicos-listagem.page.ts` e `tipo-servicos-listagem.module.ts`, que são respectivamente nossa visão, nossa classe back-end para o template e nosso módulo, que conterá os artefatos relacionados à listagem tipo de serviços.

Inicialmente, nosso código para o componente (arquivo TypeScript), deve ser semelhante ao apresentado na sequência.

```
import { Component, OnInit } from '@angular/core';

@Component({
    templateUrl: './tipo-servicos-listagem.page.html'
})
export class TipoServicosListagemPage implements OnInit {
    ngOnInit(): void {
    }
}
```

Já para o nosso arquivo HTML, o código inicial é o apresentado adiante.

```
<ion-header>
    <ion-toolbar>
      <ion-title>Tipos de serviços</ion-title>
    </ion-toolbar>
</ion-header>

<ion-content padding-left padding-top>
</ion-content>
```

Finalizando o código inicial para os três arquivos, temos o código para nosso módulo.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { Routes, RouterModule } from '@angular/router';

import { IonicModule } from '@ionic/angular';

import { TipoServicosListagemPage } from './tipo-servicos-listagem.page';

const routes: Routes = [
  {
    path: '',
    component: TipoServicosListagemPage
  }
];

@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    IonicModule,
    RouterModule.forChild(routes)
  ],
  declarations: [TipoServicosListagemPage]
})
export class TipoServicosListagemPageModule {}
```

Já sabemos que, ao executarmos nossa aplicação, a página que será exibida é a `Home`. Mas sabemos como corrigir isso, certo? Vamos abrir o arquivo `app-routing.module.ts` e atualizar a rota `home`, no `loadChildren`, com:

```
 './pages/tipo-servicos/listagem/tipo-servicos-
listagem.module#TipoServicosListagemPageModule'
```

Se você quiser agora testar sua aplicação, pode fazê-lo, mas saiba que nada será exibido ainda, a não ser o título da página. Lembre-se de que na primeira execução será solicitada a instalação do `@ionic/lab`. Caso você queira, podemos realizar a instalação com o código a seguir antes de executar a aplicação.

```
npm install @ionic/lab@2.0.1
```

4.2 Populando a página com itens

Sempre que desenvolvemos uma aplicação que manipula dados informados pelo usuário, normalmente fazemos uso de algum mecanismo de persistência, para que estes dados estejam disponíveis entre as sessões de uso da aplicação. No capítulo anterior, nós trabalhamos apenas com a interface com o usuário, que solicitava estes dados, não os armazenamos. Neste capítulo nós trabalharemos com a exibição destes dados, mas ainda sem tê-los persistidos fisicamente. Faremos uso de arrays, pois nosso foco aqui, ainda é a interface com o usuário.

Precisamos, em nosso componente, criar e disponibilizar uma matriz com os dados que queremos exibir ao usuário. No código a seguir, veja a matriz que devemos implementar, logo no início da classe. Observe o modificador de escopo `public`, que possibilitará o consumo desta variável por nosso template. Fique à vontade para inserir mais itens para ver a página totalmente ocupada.

```
public tiposServicos = [
    { id: 1, nome: 'Alinhamento', valor: 12.34 },
    { id: 2, nome: 'Balanceamento', valor: 56.78 },
    { id: 3, nome: 'Revisão freios', valor: 90.12 },
    { id: 4, nome: 'Suspensão', valor: 34.56 }
];
```

Com a fonte de dados criada, precisamos agora renderizar estes dados em nosso template e faremos isso com o `<ion-list>`, que utilizamos no capítulo anterior, para exibir os erros no início da página. Veja o código a seguir. Observe a junção de componentes Ionic com HTML e CSS.

```
<ion-content padding>
  <ion-list>
    <ion-item *ngFor="let tipoServico of tiposServicos">
        <ion-label>
            <h2>{{tipoServico.nome}}</h2>
            <p style="text-align:right">R$ {{tipoServico.valor}}</p>
        </ion-label>
    </ion-item>
  </ion-list>
</ion-content>
```

A figura a seguir apresenta os pseudoemuladores com a aplicação em execução. Nesta figura trago mais itens do que temos registrado em `tiposServicos`, ok?

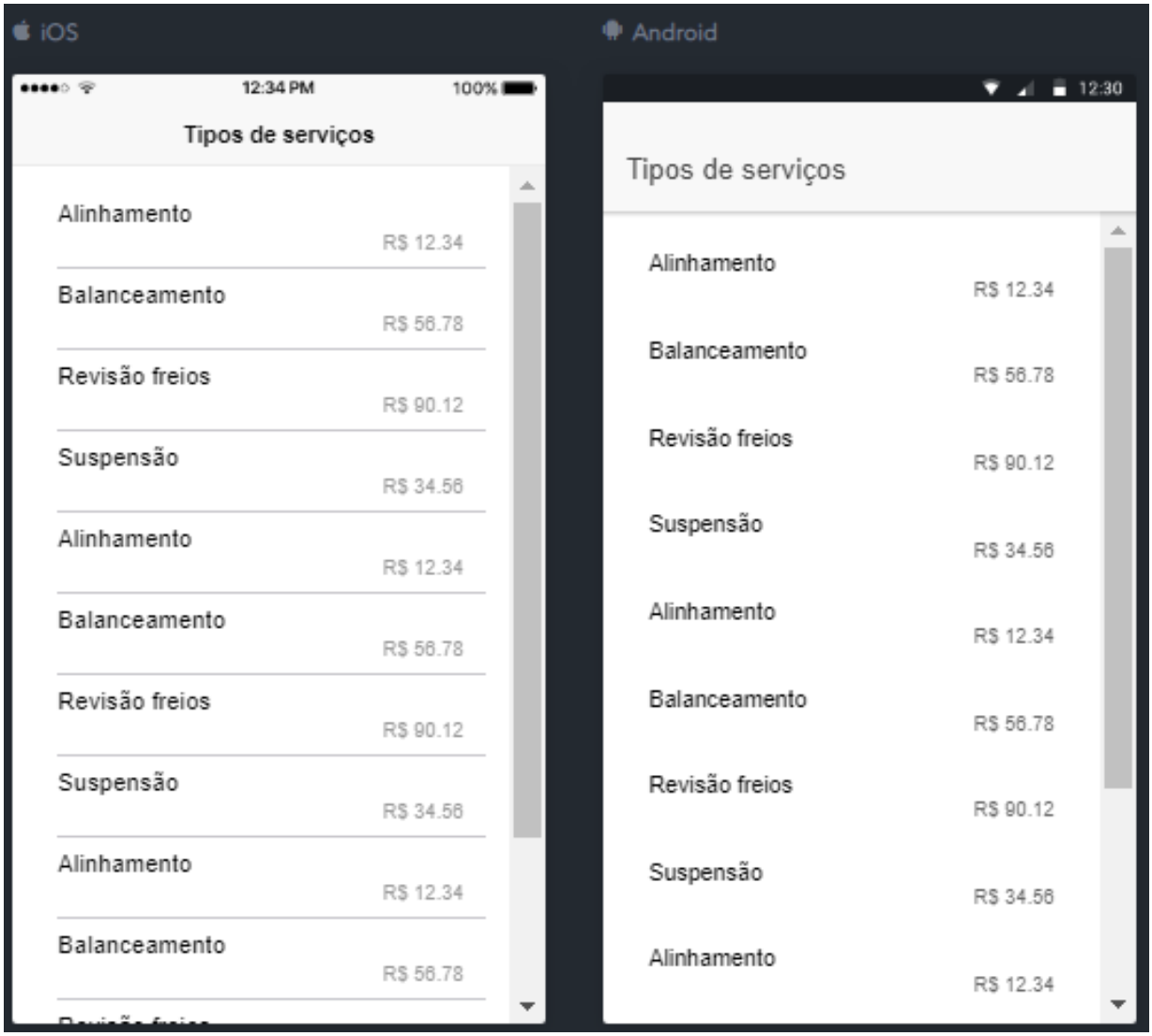


Figura 4.1: Listagem de tipos de serviços

4.3 Navegação para uma página de detalhes de um tipo de serviço

É muito comum que, de uma visão onde os dados são listados, seja possível a navegação para uma página com maiores detalhes sobre um determinado item, selecionado em tal listagem.

Da mesma maneira que criamos a página para listagem, vamos criar nossa página de inserção e alteração. Daremos aos artefatos um nome que semanticamente abrangerá a inserção e alteração de um tipo de serviço. Sendo assim, na pasta `tipo-servicos`, crie uma nova pasta, chamada `add-edit` e, dentro dela, crie os arquivos: `tipo-servicos-add-edit.page.html`, `tipo-servicos-add-edit.page.ts` e `tipo-servicos-add-edit.module.ts`. Na sequência, o código inicial para os arquivos.

Arquivo tipo-servicos-add-edit.page.html

No código a seguir, observe que temos um novo controle, o `<ion-item-divider>`. Este controle renderiza um tipo de container na página, destacando os controles que estão dentro dele. Fazemos uso do sistema de grids do Ionic no `<ion-item-divider>` e, em nossa primeira linha, temos uma coluna com uma imagem, por meio do componente `<ion-img>`. Observe o caminho para a figura. A pasta `assets` está no mesmo nível de `app` e você precisará criar a `imgs` e copiar a figura (que pode ser a que quiser), para esta pasta.

Ao final, temos dois `<ion-item>`, que exibirão os valores do tipo de serviço selecionado na listagem, da página anterior. Estes valores serão recuperados por meio do binding das `{{}}`.

```
<ion-header>
    <ion-toolbar>
        <ion-title>Tipos de serviços</ion-title>
    </ion-toolbar>
</ion-header>

<ion-content padding>
    <ion-item-divider color="light">
        <ion-grid no-padding>
            <ion-row no-padding>
                <ion-col col-3 align-self-center>
                    <ion-img style="width: 50px; height: 50px;"
src="assets/imgs/tab_servicos.png"></ion-img>
                </ion-col>
```

```
                <ion-col col-9 align-self-center style="font-size: 20px">
                    <ion-label style="font-size: 30px">Dados do
serviço</ion-label>
                </ion-col>
            </ion-row>
        </ion-grid>
    </ion-item-divider>
    <ion-item>{{tipoServico.nome}}</ion-item>
    <ion-item>
        <span item-end>R$ {{tipoServico.valor}}</span>
    </ion-item>
</ion-content>
```

Arquivo tipo-servicos-add-edit.page.ts

O código para o componente pode ser verificado na listagem a seguir. Ele ainda não declara o objeto `tipoServico`, que estamos utilizando no template da listagem anterior, mas logo veremos isso.

```
import { Component, OnInit } from '@angular/core';

@Component({
    templateUrl: './tipo-servicos-add-edit.page.html'
})
export class TipoServicosAddEditPage implements OnInit {
    ngOnInit(): void {
    }
}
```

Arquivo tipo-servicos-add-edit.module.ts

Da mesma maneira que a listagem anterior, a seguinte traz apenas o básico para a definição de nosso módulo para a página de inserção e edição de um tipo de serviço.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { IonicModule } from '@ionic/angular';
import { RouterModule, Routes } from '@angular/router';
```

```
import { TipoServicosAddEditPage } from './tipo-servicos-add-edit.page';

const routes: Routes = [
    { path: '', component: TipoServicosAddEditPage }
  ];

@NgModule({
    imports: [
      CommonModule,
      FormsModule,
      IonicModule,
      RouterModule.forChild(routes),
      ReactiveFormsModule
    ],
    declarations: [
      TipoServicosAddEditPage
    ]
})

export class TipoServicosAddEditPageModule {}
```

Precisamos agora implementar a navegação da página de listagem para a nova que criamos. Nosso primeiro passo é definir no `app-routing.module.ts` a rota a seguir.

```
{ path: 'add-edit/:id', loadChildren: './pages/tipo-servicos/add-edit/tipo-servicos-add-edit.module#TipoServicosAddEditPageModule' }
```

No código anterior, observe que definimos a rota recebendo um argumento, que será o `id` do tipo de serviço selecionado na listagem e que deverá ser recuperado pela nova página, para então ter seus dados exibidos.

Mas como chegar à rota especificada anteriormente? É bem simples, vamos apenas adicionar no `<ion-item>` da listagem (no template) a instrução a seguir. Observe o binding para `tipoServico`, que é a variável inicializada a cada iteração do `*ngFor`. Isso deve ser atualizado no arquivo `tipo-servicos-listagem.page`.

```
<ion-item *ngFor="let tipoServico of tiposServicos" routerLink='/add-edit/{{tipoServico.id}}'>
```

Já podemos testar nossa aplicação. Na listagem, selecione um tipo de serviço. Você será redirecionado para a página de detalhes e verá que nossos dados ainda não aparecem. Logo veremos isso. Temos agora um novo problema: como retornar para a página anterior? Em nosso template para a página de detalhes, logo no início, dentro do `<ion-toolbar>` e antes do `<ion-title>`, insira o código a seguir, lembrando que é no arquivo `tipo-servicos-add-edit.page.html`.

```
<ion-buttons slot="start">
    <ion-back-button defaultHref=""></ion-back-button>
</ion-buttons>
```

No código anterior, temos o surgimento de `slot`, que é uma característica relacionada a `Web Components`. O `slot` refere-se ao posicionamento de um conteúdo dentro de um contêiner, que em nosso caso é a barra de tarefas. Um link simples e direto sobre este tema é https://www.joshmorony.com/understanding-how-slots-are-used-in-ionic-4/, que você pode ler no momento em que quiser.

Voltando ao código, observe o componente `<ion-back-button>`, que define como URL de destino uma string vazia que, se você retornar ao `app-routing.module.ts`, verá que é uma rota que redireciona a requisição para nosso `home`, que se refere à nossa listagem. Vamos testar novamente nossa aplicação?

4.4 Recuperação do tipo de serviço selecionado

Já temos nossa página de listagem com os itens apontando para uma página de detalhes e enviando o `id` do tipo de serviço que deve ser exibido, mas, como exibir os dados específicos para o `id` recebido por esta página?

Nossa listagem é populada por um objeto que representa uma matriz e que está definida no componente de listagem, então, nossa página de detalhes não conseguirá pesquisar essa matriz, a não ser que a enviemos como parâmetro para a página, mas não é isso que faremos. Faremos uso de serviços (*services*) e injeção de dependência (*dependency injection*).

O Angular e, consequentemente, o Ionic oferecem um tipo de componente chamado `Services`, que são classes que podem ter objetos injetados em nossos componentes. Para nosso problema, temos a situação de que precisamos de um conjunto de dados, que no momento é uma matriz, mas poderia ser uma base de dados ou dados fornecidos por serviços REST.

Vamos então trazer para nossa aplicação a criação de um serviço que oferecerá acesso aos dados que deverão ser consumidos. O primeiro passo é na pasta `app` criarmos uma nova pasta chamada `services` e, nela, um arquivo chamado `tipo-servicos.service.ts`. Com o arquivo criado, vamos implementar nele o seguinte código.

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class TipoServicosService {

  private tiposServicos = [
    { id: 1, nome: 'Alinhamento', valor: 12.34 },
    { id: 2, nome: 'Balanceamento', valor: 56.78 },
    { id: 3, nome: 'Revisão freios', valor: 90.12 },
    { id: 4, nome: 'Suspensão', valor: 34.56 }
  ];

  constructor() { }

  getById(id: number) {
    const tipoServicoSelecionado = this.tiposServicos.filter(
      tipoServico => tipoServico.id === id
```

```
    );
    return tipoServicoSelecionado[0];
  }
}
```

Note que o serviço nada mais é que uma classe comum, que em nosso caso será visto também como um `provider`, por isso a existência de um decorator chamado `@Injectable`. Como nossa intenção é prover este serviço para toda a aplicação, enviamos como argumento para o decorator o elemento `providedIn` com o valor `root`. Isso gerará uma única instância do serviço, a ser compartilhada por toda a aplicação. Se fôssemos prover o serviço para um módulo específico, iríamos declará-lo em sua especificação.

Observe também a declaração de nossa já conhecida matriz, pois queremos que os dados agora sejam fornecidos por uma única fonte. Após o construtor, temos a definição de um método, que receberá um `id`. Este valor será utilizado para filtrar um elemento que coincida em valor e tipo (`===`) com a propriedade `id` de um elemento da matriz. Ao final, retornamos o elemento selecionado. Caso nada seja encontrado, `undefined` é atribuído ao elemento.

Apenas para fins de conhecimento, outra maneira de criar o componente `service` seria por meio do Ionic CLI, com a instrução `ionic generate service services/tipo-servicos.service`.

Precisamos agora ajustar nosso componente `tipo-servicos-add-edit.page.ts` para receber nosso serviço e então poderemos consumir o método que implementamos nele. Veja o novo código para nosso componente na sequência.

```
import { ActivatedRoute } from '@angular/router';
import { TipoServicosService } from '../../../services/tipo-
servicos.service';
import { OnInit, Component } from '@angular/core';

@Component({
    templateUrl: './tipo-servicos-add-edit.page.html'
```

```
})
export class TipoServicosAddEditPage implements OnInit {
    public tipoServico;

    constructor(
        private route: ActivatedRoute,
        private tipoServicoService: TipoServicosService
        ) {}

    ngOnInit(): void {
        const id: number = Number(this.route.snapshot.paramMap.get('id'));
        this.tipoServico = this.tipoServicoService.getById(id);
    }
}
```

A primeira instrução refere-se à importação de nosso componente de serviço que deve ser adicionado aos já existentes. O decorator se mantém como está. Antes do construtor, há a definição de uma propriedade chamada `tipoServico`. Ela conterá o tipo de serviço selecionado na página de listagem e que terá seus dados consumidos pelo template de exibição dos detalhes que estamos implementando. Na sequência, temos nosso construtor, recebendo um objeto para tratamento de rotas (`ActivatedRoute`) e nosso serviço anteriormente implementado (`TipoServicosService`).

Finalizando o código, temos a implementação para o `ngOnInit()`, que recupera o parâmetro enviado pela página de listagem e o utiliza para recuperar o objeto completo, que é então atribuído à nossa propriedade, para que os dados possam ser exibidos corretamente pelo template. Não nos preocupamos em verificar se um dado foi recuperado corretamente, pois, em nosso caso, o dado foi exibido, então temos a garantia de sua existência, mas é algo em que devemos pensar, talvez verificando e validando o objeto retornado, ou trabalhando com exceções, no método de serviço - veremos isso adiante. Vamos agora ver nossa página de detalhes de um tipo de serviço devidamente renderizada.

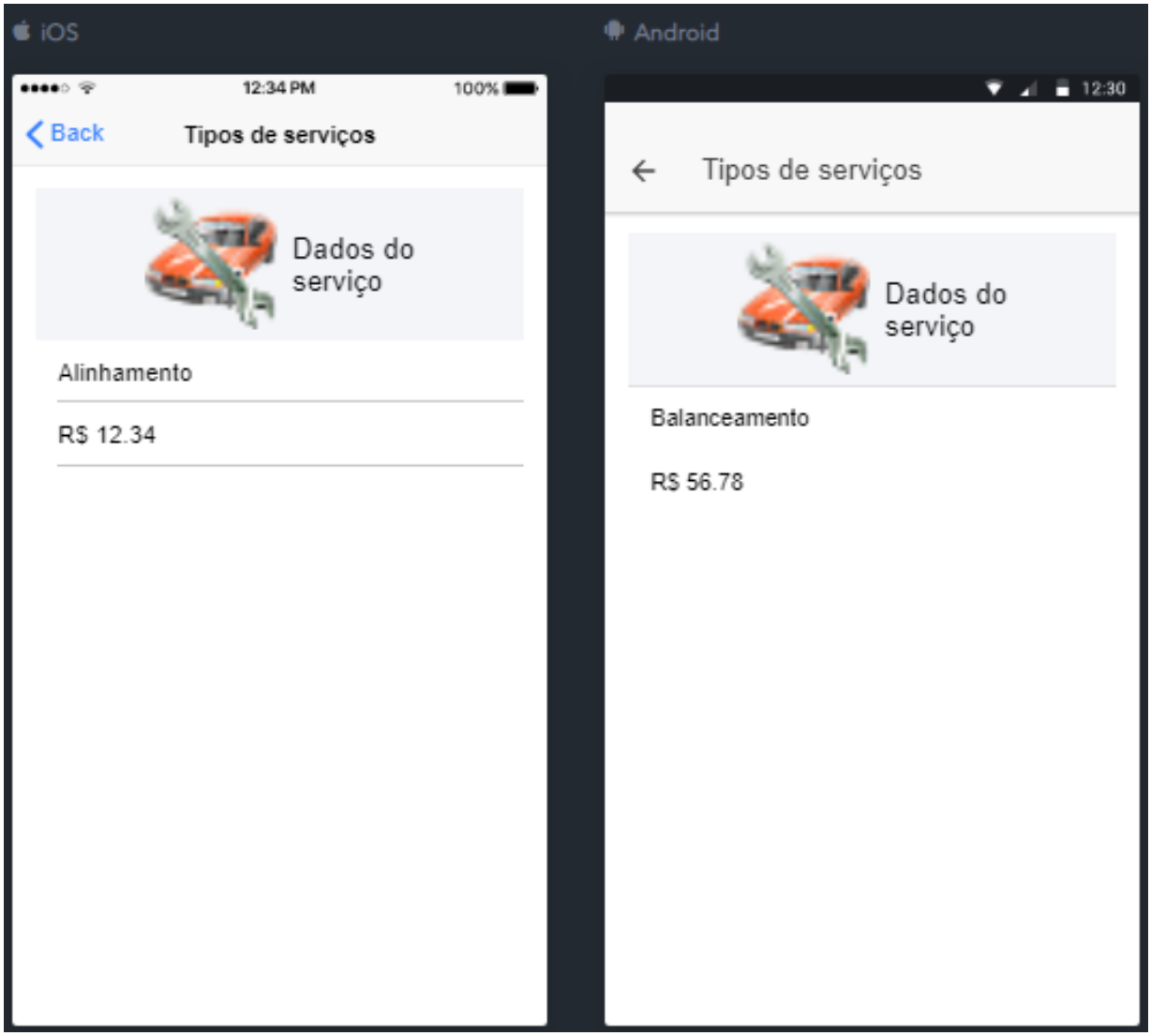


Figura 4.2: Detalhes de um item selecionado na listagem

4.5 Alteração dos dados exibidos na página de detalhes

Já temos nossa página de detalhes exibindo corretamente os dados de um tipo de serviço selecionado pelo usuário em uma página de listagem. Precisamos agora oferecer ao nosso usuário a possibilidade de alterar estes dados. Faremos uso de um botão para

permitir esta funcionalidade e teremos algumas mudanças também em nosso código de componente e template. Nosso primeiro passo será a inserção do botão que habilitará a alteração. Veja o código a seguir, que deve ser inserido no arquivo `tipo-servicos-add-edit.page.html` antes da tag de fechamento `<ion-content>`.

```
<ion-button *ngIf="!modoDeEdicao" shape="round" color="primary"
expand="block" padding (click)="iniciarEdicao()">
    Alterar dados
</ion-button>
```

O componente relativo a botões não é novo para nós, já o vimos nos capítulos anteriores. Tirando os atributos de configuração, temos a diretiva condicional do Angular, `*ngIf`, que determinará quando o controle será renderizado e podemos verificar que isso ocorrerá na negação de uma variável, chamada `modoDeEdicao`, que já declararemos. Ao final da tag `<ion button>` de abertura, temos um `Event Bind` para `click`, para um método que também não implementamos ainda.

Em nosso arquivo do componente inclua, antes da assinatura do construtor, a instrução a seguir, e, após o construtor, o método que também está na listagem.

```
// Declaração de propriedade para sinalizar modo de edição e consulta
public modoDeEdicao = false;

// Método que registrará que o tipo de serviço estará em processo de
edição
iniciarEdicao() {
    this.modoDeEdicao = true;
}
```

Execute sua aplicação e veja a exibição do botão, clique/pressione-o e veja que o botão desaparece. É este caminho que queremos seguir.

Utilizamos o binding por meio de `{{}}` com nosso componente para então termos o dado exibido ao usuário. Entretanto, agora não

queremos essa página apenas para exibir dados, queremos nela a possibilidade de o usuário alterá-los. Isso nos leva à necessidade de um formulário e componentes que renderizem controles de interação. Precisamos reformular nosso template, tal qual segue na listagem para o arquivo `tipo-servicos-add-edit.page.html`, abaixo de `</ion-item-divider>`, substituindo o código existente.

```
<form [formGroup]="tiposServicosForm" (submit)="submit()">
    <ion-item lines="none" [disabled]="!modoDeEdicao">
        <ion-label position="floating" class="label-input-
title">Nome</ion-label>
        <ion-input formControlName="nome" type="text" style="margin-left:
3%;"></ion-input>
    </ion-item>

    <ion-item lines="none" [disabled]="!modoDeEdicao">
        <ion-label position="floating" class="label-input-
title">Valor</ion-label>
        <ion-input text-right formControlName="valor" type="number"></ion-
input>
    </ion-item>

    <ion-button *ngIf="!modoDeEdicao" shape="round" color="primary"
expand="block" padding (click)="iniciarEdicao()">
        Alterar dados
    </ion-button>

    <div *ngIf="modoDeEdicao" no-padding>
        <ion-grid>
            <ion-row>
                <ion-col col-6 >
                    <ion-button shape="round" color="success"
                                size="small" padding expand="block"
                                type="submit">
                        Gravar
                    </ion-button>
                </ion-col>

                <ion-col col-6>
                    <ion-button shape="round" color="warning"
```

```
                                size="small" padding expand="block"
                                (click)="cancelarEdicao()">
                        Cancelar
                    </ion-button>
                </ion-col>
            </ion-row>
        </ion-grid>
    </div>
</form>
```

Logo no início do código anterior, temos a definição de nosso formulário, que é um Reactive Form. Temos um nome ligado à propriedade `formGroup` e um evento ligado ao `submit`. Depois, dois `<ion-item>` apresentando os dados do tipo de serviço selecionado, já com controles que possibilitem a alteração dos dados. Mas atenção: veja nos `<ion-item>` a definição e ligação do `disabled` com nossa variável de controle. Com isso, os controles só aceitarão interação quando clicarmos no botão para alterar os dados.

Após o botão que habilita a alteração de dados, temos uma `<div>`, que será exibida apenas quando estivermos alterando um tipo de serviço, pois ela contém dois botões: um que realizará a gravação e será o botão de submissão do formulário, e um que registrará o cancelamento dos dados alterados. Veja o uso de grids para os botões.

Precisamos agora declarar em nosso arquivo do módulo (TypeScript) a importação de `ReactiveFormsModule`. Depois, precisamos definir nosso `FormGroup`, que estamos usando em nossa tag `<form>`. E essa é a instrução da listagem a seguir, que deve ser implementada antes do construtor. Será preciso importar `FormGroup`, mas o Visual Studio Code ajuda você nisso.

```
public tiposServicosForm: FormGroup;
```

Dando sequência, na preparação de nosso componente para o Reactive Form, precisamos injetar a instrução a seguir em nosso construtor. Precisaremos também importar `FormBuilder`.

```
private formBuilder: FormBuilder
```

Vamos à adaptação de nosso método `ngOnInit()`, tal qual segue. As duas primeiras instruções já temos implementadas, mas preferi mantê-las na listagem. Observe que após termos recuperado o objeto selecionado na listagem, configuramos nosso `FormGroup`, criando suas propriedades e atribuindo a elas o valor inicial, com base no objeto recuperado. Definimos também uma validação básica. Atente para a importação para `Validators`.

```
ngOnInit(): void {
    const id: number = Number(this.route.snapshot.paramMap.get('id'));
    this.tipoServico = this.tipoServicoService.getById(id);

    this.tiposServicosForm = this.formBuilder.group({
        id,
        nome: [this.tipoServico.nome, Validators.required],
        valor: [this.tipoServico.valor, Validators.required]
    });
}
```

Execute a aplicação, selecione um item da listagem, veja os controles renderizados e desabilitados. Clique/pressione no botão que habilita a alteração, observe a ocultação deste botão e exibição dos outros dois, assim como a habilitação dos controles de interação. Compare sua aplicação com a figura a seguir.

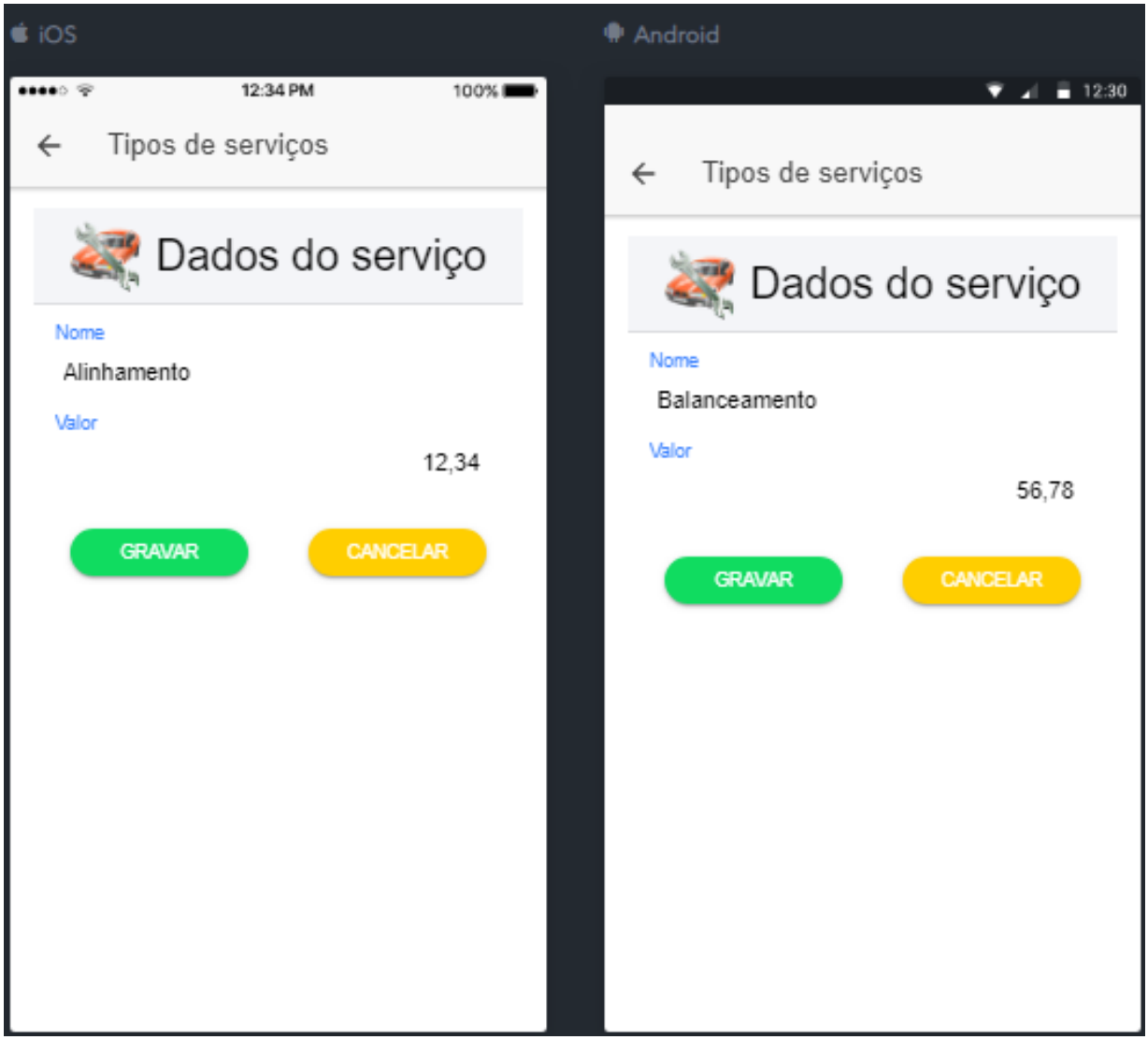


Figura 4.3: Alteração de um tipo de serviço

Com a interface com o usuário pronta, precisamos realizar a atualização dos dados alterados. Em nosso código anterior, para o template, na tag `<form>`, lembra que fizemos um binding com `submit`? Pois é, precisamos agora implementar nossos serviços que deverão ser atendidos por essa operação. Começaremos com a implementação dos métodos em nossa classe de serviço, onde implementamos nossa matriz e o método `getById()`. Veja os novos métodos, no arquivo `tipo-servicos.service.ts`.

```
private getIndexOfElement(id: number): number {
    return this.tiposServicos.indexOf(this.getById(id));
}

update(tipoServico: any) {
    this.tiposServicos[this.getIndexOfElement(tipoServico.id)] =
tipoServico;
}
```

O primeiro método da listagem anterior nos retornará o índice do elemento referente ao `id` que o método recebe. Como estamos lidando com matrizes, para atualizarmos, este é o caminho. Já o segundo método, `update()`, atualizará o elemento com índice recuperado pelo método `getIndexOfElement()`. Agora sim, vamos ao método `submit()`, em nosso arquivo TypeScript da página de edição. Veja o código na sequência.

```
submit() {
    this.tipoServicoService.update(this.tiposServicosForm.value);
    this.modoDeEdicao = false;
}
```

Observe que enviamos o valor atual de nosso `FormGroup` para o método `update()` de nosso serviço, pois os dados informados pelo usuário estão nos componentes deste objeto. Após a execução da atualização, alteramos o estado de nossa página para apenas visão, por meio de nossa variável controladora `modoDeEdicao`.

Agora, já podemos realizar a alteração e gravar os novos dados. Na aplicação, após gravar os novos dados, volte para a listagem. Verifique que ela não mostra a atualização registrada.

Abra o TypeScript de nosso componente de listagem. Veja que, no início deste capítulo, tínhamos definido uma matriz e que o nosso template está baseado nesta coleção. Você se lembra de que, para disponibilizarmos os dados do item selecionado na listagem, criamos um serviço e disponibilizamos neste serviço uma matriz que será de uso comum em toda a aplicação? Pois bem, estamos assim exibindo uma coleção na listagem e trabalhando em cima de outra

coleção na alteração. Retire a matriz e deixe apenas a declaração a seguir no componente de listagem.

```
public tiposServicos;
```

Inicializaremos a propriedade anterior com base em dados obtidos em nosso serviço, o que nos leva a injetar este serviço em nosso componente. Desta maneira, insira a seguinte declaração no construtor de nosso componente. Precisaremos importá-lo.

```
private tipoServicoService: TipoServicosService
```

Agora precisamos trabalhar em nosso serviço, pois ele não nos oferece nada para recuperar todos os tipos de serviços existentes. Sendo assim, no arquivo `tipo-servicos.service.ts` , adicione o seguinte método.

```
getAll() {
    return this.tiposServicos;
}
```

Vamos agora retornar para nosso componente de alteração de páginas e consumir este serviço, e faremos isso, como já de costume para processos relacionados à inicialização do componente, no método `ngOnInit()` . Veja a implementação a seguir, que já se traduz por si só. O arquivo a implementar é o `tipo-servicos-listagem.page.ts` .

```
ngOnInit() {
    this.tiposServicos = this.tipoServicoService.getAll();
}
```

Execute sua aplicação, selecione um item na listagem, altere-o, grave a alteração e retorne para a listagem. Veja que ela agora se apresenta atualizada, pois a coleção de objetos é a mesma, para listagem e alteração.

Resta-nos agora a implementação para o botão que cancela o processo de alteração. Veja esta implementação a seguir, que deve ser realizada no arquivo `tipo-servicos-add-edit.page.ts` . Note que

atribuímos ao `FormGroup` o objeto inicial, recuperado na construção do componente, cancelando qualquer alteração realizada nos controles e retornando nossa variável de controle ao estado de visualização.

```
cancelarEdicao() {
    this.tiposServicosForm.setValue(this.tipoServico);
    this.modoDeEdicao = false;
}
```

4.6 Navegação para uma página de inserção de dados utilizando classes de negócio

Em uma página onde os dados são listados, é muito comum ser possível navegar para uma página de inserção de novos dados para a listagem em foco. Vamos fazer uso de um controle Ionic, chamado `FAB` (*Floating Action Button*), para que o usuário possa realizar esta navegação. Vamos inseri-lo em nosso template de listagem, antes do fechamento da tag `<ion-content>` . Veja o código a seguir.

```
<ion-fab vertical="top" horizontal="end" slot="fixed" edge="true">
    <ion-fab-button routerLink='/add-edit/-1'>
        <ion-icon name="add"></ion-icon>
    </ion-fab-button>
</ion-fab>
```

Execute sua aplicação e note, na página de listagem, no topo (`top`) à direita (`end`), às margens da listagem e cabeçalho (`edge`), a inserção de um botão em forma de círculo, com o ícone de adição (`add`). Notou as diretivas e valores durante a sentença anterior, entre parênteses? Observou também o valor negativo enviado na rota? Compare sua visão com a da figura a seguir.

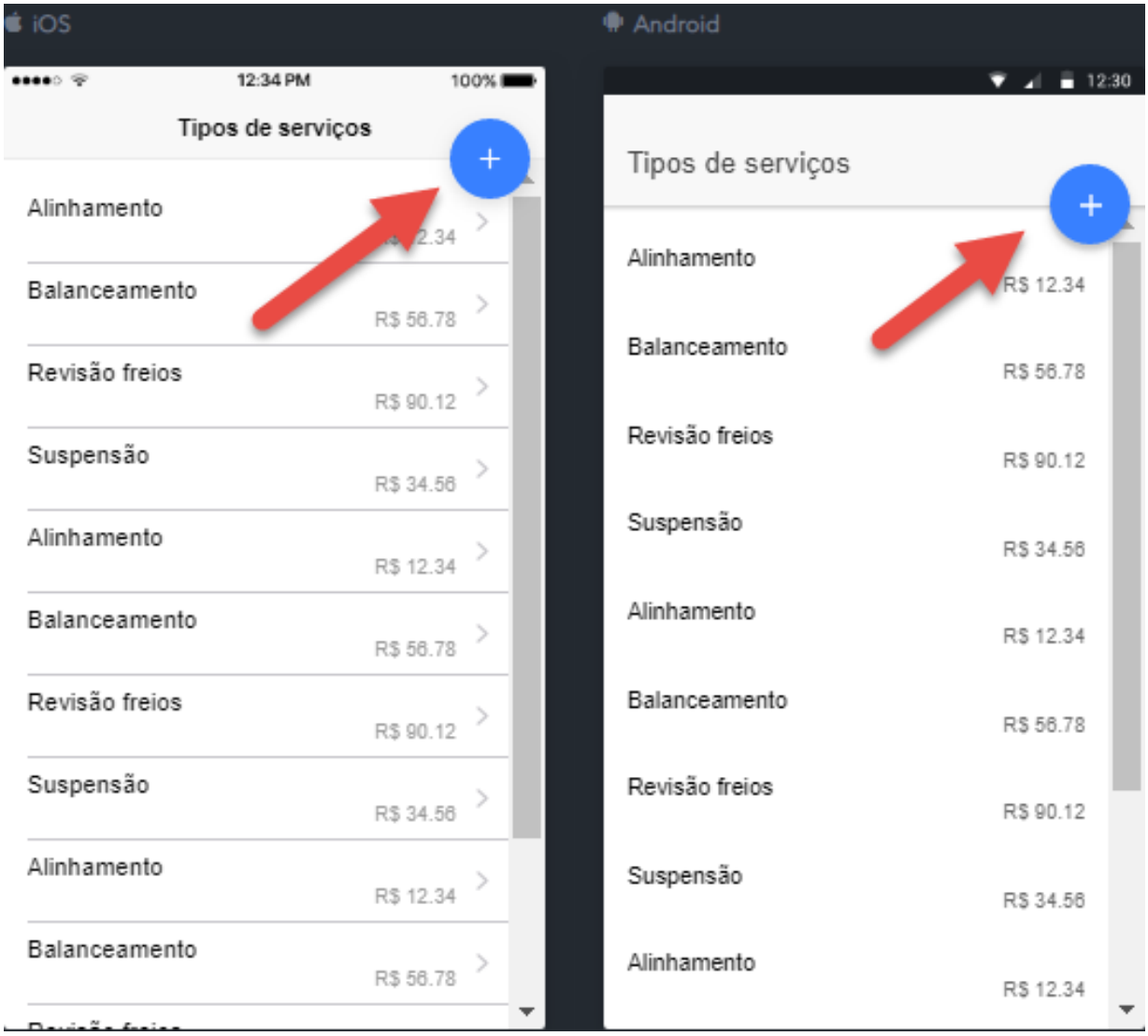


Figura 4.4: FAB para adicionar novos valores

Vamos começar a implementação da funcionalidade para o FAB trazendo para nosso projeto o uso de classes de modelo. Em TypeScript, quando temos um modelo apenas de propriedades, a recomendação é o uso de `interface`, em vez de `class`. Vamos então criar, na pasta `app`, uma nova pasta, chamada `models`. Dentro dela, crie um arquivo chamado `tipo-servico.model.ts` e, dentro dele, insira o código a seguir.

```
export interface TipoServico {
    id: number;
    nome: string;
```

```
    valor: number;
}
```

Observe que estamos criando a interface de maneira semelhante à qual criamos as classes de nossos componentes, apenas utilizando a palavra reservada `interface`.

A utilização de modelos de negócio, princípio da Orientação a Objetos, nos permitirá tipificar nossos objetos, arrays e coleções, que até então aceitam `any` como tipo de dados. Com esta nova técnica, precisamos, em nosso serviço, tipificar o objeto array, e podemos fazer isso alterando a instrução de declaração da matriz para como está exibido na sequência. Na listagem também está a nova assinatura para alguns métodos, também da classe de serviço (`tipo-servicos.service.ts`).

```
// Declaração da array
private tiposServicos: TipoServico[] = [
    { id: 1, nome: 'Alinhamento', valor: 12.34 },
    { id: 2, nome: 'Balanceamento', valor: 56.78 },
    { id: 3, nome: 'Revisão freios', valor: 90.12 },
    { id: 4, nome: 'Suspensão', valor: 34.56 }
];

// Assinatura de métodos com tipo de retorno
getById(id: number): TipoServico

getAll(): TipoServico[]

// Tipificação do argumento recebido
update(tipoServico: TipoServico)
```

Precisamos também alterar a declaração de nossa variável que armazenará o tipo de serviço em tela. Veja na sequência. Observe também que já aproveitamos e o tornamos privado, pois não o utilizaremos mais em nosso template.

```
private tipoServico: TipoServico;
```

Precisamos pensar no fato de que nosso componente e template são responsáveis pela visualização do item selecionado na listagem, em sua alteração, e agora em inserção de um novo item. Onde temos os dados a serem exibidos na página recuperados? No método `ngOnInit()` do componente (`tipo-servicos-add-edit.page.ts`). Vamos adaptá-lo para o código a seguir.

```
ngOnInit() {
    const id: number = Number(this.route.snapshot.paramMap.get('id'));

    if (id > 0) {
        this.tipoServico = this.tipoServicoService.getById(id);
    } else {
        this.tipoServico = {id, nome: '', valor: 0.00 };
        this.modoDeEdicao = true;
    }

    this.tiposServicosForm = this.formBuilder.group({
        id: [this.tipoServico.id],
        nome: [this.tipoServico.nome, Validators.required],
        valor: [this.tipoServico.valor, Validators.required]
    });
}
```

Veja que, tirando o bloco `if...else` , o resto já conhecemos. A diferença está no momento de inicializarmos a propriedade `tipoServico` , que, até antes do código anterior, era inicializada apenas pela recuperação de um item já existente em nosso serviço. Agora isso mudou, pois temos a situação de envio de um parâmetro com valor `-1` , que sinalizará para nossa aplicação que queremos uma interface de inserção. Por isso, nosso `else` tem as instruções de inicialização da propriedade e alteração de nosso flag de edição, a `modoDeEdicao` .

Mas e quando formos gravar? O id negativo precisará receber um valor válido, que, para nossa lógica simples, será o valor seguinte ao último item de nosso array. E faremos isso no momento da atualização. Veja, em nossa classe de serviço (`tipo-servicos.service`), o novo código para o `update()` .

```
update(tipoServico: TipoServico) {
    if (tipoServico.id < 0) {
      tipoServico.id = this.tiposServicos[this.tiposServicos.length -
1].id + 1;
      this.tiposServicos.push(tipoServico);
    } else {
      this.tiposServicos[this.getIndexOfElement(tipoServico.id)] =
tipoServico;
    }
}
```

Veja que fazemos também uso de um `if...else` para que o método saiba o que deve fazer com o objeto recebido, ou seja, é preciso inserir, ou atualizar e, com base no `id`, o método saberá o que fazer. Atribuímos o valor do id para o novo objeto e então realizamos o `push()`, para que ele seja inserido na matriz.

Podemos agora concluir nossa implementação referente à inserção e alteração, exibindo ao usuário uma mensagem sobre a realização do processo e um redirecionamento dele para a página de listagem.

No capítulo anterior, vimos algumas possibilidades para exibição de mensagens ao usuário e vamos optar agora pelo `Toaster`. Entretanto, em vez de criarmos em nosso componente um método que exibirá o Toaster, vamos utilizar um recurso aprendido aqui, o de *services*.

Na pasta `services`, que já temos na estrutura de nossa aplicação, crie um arquivo chamado `toast.service.ts` e, como código, implemente a listagem a seguir. Veja que o método é o mesmo que implementamos no capítulo anterior.

```
import { ToastController } from '@ionic/angular';
import { Injectable } from '@angular/core';

@Injectable({
    providedIn: 'root'
  })
export class ToastService {
```

```
    constructor(private toastCtrl: ToastController) {}

    async presentToast(message: string, duration: number, position: 'top'
| 'bottom') {
        const toast = await this.toastCtrl.create({
          message,
          duration,
          position
        });
        toast.present();
    }
}
```

Agora precisamos consumir este serviço em nosso componente de edição/inserção de um tipo de serviço. Desta maneira, no construtor da classe do arquivo `tipo-servicos-add-edit.page.ts`, insira o código a seguir como argumento.

```
private toastService: ToastService
```

Como temos um objeto `ToastService` injetado, podemos efetivamente consumi-lo e faremos isso em nosso método `submit()`, após a chamada ao `update()`, inserindo o código a seguir.

```
this.toastService.presentToast('Gravação bem sucedida', 3000, 'top');
```

Enfim, para redirecionarmos a aplicação para nossa listagem inicial, precisamos injetar também em nossa classe um objeto responsável por nossas rotas, o que nos leva a inserir o código a seguir em nosso construtor.

```
private router: Router
```

Para efetivar o redirecionamento, após a invocação ao nosso método para exibição do Toaster, precisamos implementar a instrução a seguir, que redireciona a aplicação para a rota `''`, que levará para a `home`.

```
this.router.navigateByUrl('');
```

Uma observação: como agora estamos redirecionando a aplicação após a gravação, seja ela inserção ou atualização, não precisamos, no método `submit()`, nos preocupar mais com a variável de controle de estado em edição. Por outro lado, poderíamos pensar no redirecionamento apenas quando for uma atualização e, quando estivéssemos inserindo um registro, a página fosse disponibilizada para um novo Tipo de Serviço.

4.7 Remoção de um item da lista

Estamos quase concluindo nossas operações CRUD para Tipos de Serviços, mas, você já sabe o que é CRUD? Caso não saiba, CRUD é um acrônimo para *Create, Read, Update e Delete* (Criação, Leitura, Atualização e Remoção). Isso quer dizer que, quando estamos implementando um CRUD, estamos desenvolvendo um serviço que possibilite estas quatro operações básicas para um determinado tipo de dado.

Até aqui temos a criação (inserção), leitura (busca por todos os registros e pelo id), atualização (alteração), faltando apenas a remoção de dados de nossa matriz. E é este nosso objetivo nesta seção.

Em listagens nas aplicações móveis, temos um recurso intuitivo, que todos os usuários seguem, que é o de arrastar um item. Isso é conhecido como `slide`, que, traduzindo para nossa funcionalidade, seria `deslizar`, que fica melhor que o "arrastar", que em aplicações de janelas nos remete ao `Drag and Drop`.

Com isso posto, podemos abstrair que precisamos adaptar nossa listagem para possibilitar ao usuário esta funcionalidade de deslizar um item e, a partir desta operação, uma opção para remover o item em referência seja apresentada. Vamos adaptar nosso código no

arquivo HTML de listagem para o `<ion-list>`. Veja o novo código na sequência.

```
<ion-list #slidingList>
    <ion-item-sliding *ngFor="let tipoServico of tiposServicos">
      <ion-item routerLink='/add-edit/{{tipoServico.id}}'>
        <ion-label>
          <h2>{{tipoServico.nome}}</h2>
          <p style="text-align:right">
            R$ {{tipoServico.valor}}</p>
        </ion-label>
      </ion-item>

      <ion-item-options side="end">
        <ion-item-option color="danger"
(click)="removerTipoServico(tipoServico)">
          <ion-button color="danger" no-padding>
              <ion-icon name="trash"></ion-icon>
              Remover
          </ion-button>
        </ion-item-option>
      </ion-item-options>
    </ion-item-sliding>
</ion-list>
```

Verifique, após o `<ion-list>`, o uso de um novo componente, o `<ion-item-sliding>`, que tem em si a seleção dos itens que serão iterados, tirando esta funcionalidade do `<ion-item>`. Ainda nesta tag, veja a variável de template `#slidingList`. Ela terá a finalidade de um *workaround* para que, ao removermos um item, seja possível deslizarmos outro.

Após o `<ion-item>`, temos um componente responsável pelo comportamento que desejamos quando o usuário deslizar um item da listagem. Estamos falando do `<ion-item-options>`, que define um conjunto de `<ion-item-option>` para as possíveis opções a serem oferecidas ao usuário, após o deslizamento do item. Em nosso caso teremos apenas a de remoção do item da coleção que popula a

listagem, o que nos leva a um único `<ion-item-option>`, que tem nele a definição de um `<ion-button>` com um `<ion-icon>`.

Observe que no `<ion-item-option>` temos um *event binding* com `removerTipoServico()`, que receberá o serviço atual, exibido no item deslizado da listagem. Logo implementaremos este método.

Observe na tag `<ion-item-options>` a definição do atributo `side` com `"end"`, o que significa que as opções para cada item estarão exibidas ao lado direito do item deslizado. Outra opção seria `start`, que os colocaria ao lado esquerdo.

Já para a exibição do ícone tradicional de lixeira no botão, estamos dando ao `<ion-icon>`, na propriedade `name`, o valor `trash`, o que refletirá no ícone desejado. Estes nomes e figuras podem ser localizados em https://ionicframework.com/docs/ionicons/. Veja a figura a seguir com a exibição destas opções.

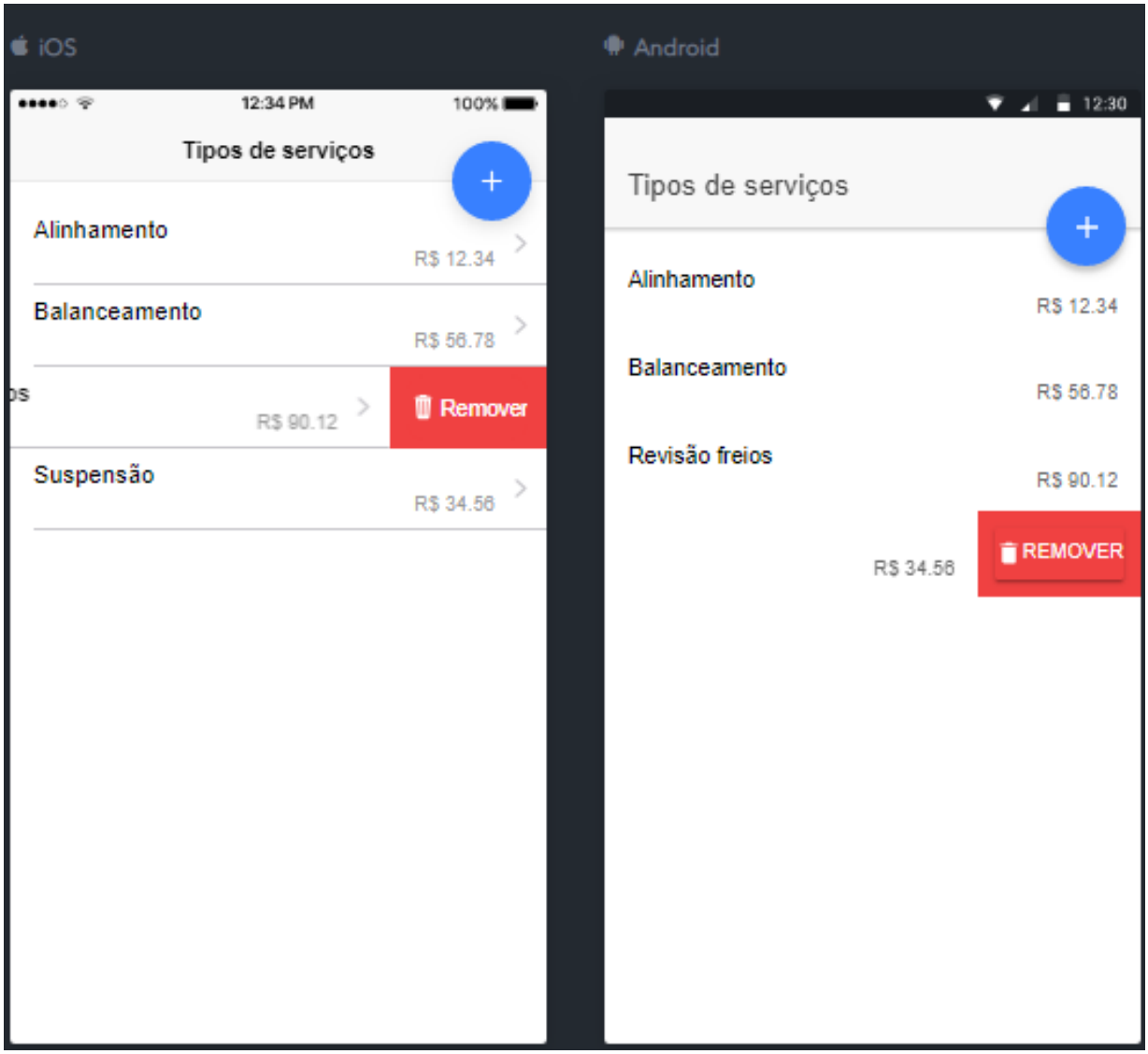


Figura 4.5: Deslizando um item da listagem

Precisamos agora implementar nosso componente, para que as implementações que fizemos no template possam ter seu comportamento capturado e executado de acordo ao esperado. Nosso primeiro passo é recuperar nossa variável de template `slidingList`, que definimos no código anterior. Para isso, antes do construtor, insira o código a seguir.

```
@ViewChild('slidingList') slidingList: IonList;
```

Precisamos criar o nosso método `removerTipoServico()`, que será invocado quando o usuário confirmar o desejo de remoção do item deslizado. Veja o código para este método na listagem a seguir. Observe a definição do método como assíncrono, pois nossa última instrução é `await`, e é ela que permitirá que após a remoção de um item, outros deslizes possam ser realizados.

```
async removerTipoServico(tipoServico: TipoServico) {
    this.tipoServicoService.remove(tipoServico);
    this.toastService.presentToast('Tipo de serviço removido', 3000,
'top');
    await this.slidingList.closeSlidingItems();
}
```

A implementação do método anterior depende de outras antes de podermos testar nossa aplicação. A primeira refere-se a implementação do método `remove()`, em nosso `TipoServicosService`. Veja-o na sequência.

```
remove(tipoServico: TipoServico) {
    this.tiposServicos.splice(this.getIndexOfElement(tipoServico.id), 1);
}
```

A segunda alteração que precisamos realizar é em relação ao `this.toastService`, que precisamos também injetar em nosso construtor, tal qual fizemos para a página de inserção/consulta/alteração. Veja o código a seguir, que devemos inserir em nosso construtor, na classe de listagem de tipos de serviço.

```
private toastService: ToastService
```

Conclusão

Terminamos este interessante capítulo, onde realizamos toda a operação relacionada a um CRUD. Vimos vários recursos e técnicas, que poderemos utilizar em nossas aplicações futuras e certamente usaremos nos capítulos seguintes.

Recapitulando e relembrando, vimos novamente o processo de criação de um projeto, criamos uma página de listagem, populando-a com itens armazenados em uma matriz, fizemos uso de `Services`, aprendemos e praticamos a navegação entre páginas, enviando objetos para definição de comportamento após a navegação.

Foi possível inserirmos um item em nossa matriz, alterá-lo e removê-lo, mantendo sempre nossos controles visuais atualizados e com comportamentos de acordo com seu estado.

É hora de tomarmos um copo de água e partirmos para o próximo capítulo, onde veremos um pouco de persistência local.

Capítulo 5
Persistindo dados fisicamente em aplicações Ionic

No capítulo anterior, implementamos um CRUD para o modelo de negócio `Tipo de Serviço`, onde os dados eram mantidos em uma matriz, o que não garantia sua persistência entre execuções da aplicação.

Veremos neste quinto capítulo alguns mecanismos para persistência física de dados. Aplicaremos estes recursos e técnicas em modelos de negócio relacionados a peças a serem utilizadas em manutenções de veículos, com um CRUD completo, e começaremos também um CRUD com ordem de serviço registrada na entrada do veículo. Trabalharemos o armazenamento interno do Ionic, conhecido com Storage, para peças, e depois com ordens de serviço trabalharemos a metodologia mais comum, uma base de dados, com tabelas.

5.1 Criação dos artefatos de modelo e de registro

Vamos dar continuidade no modelo que trabalhamos no capítulo anterior. Vamos implementar nosso modelo de negócio e depois nossa página de registro de dados, onde persistiremos fisicamente nossos objetos de peças de veículo.

Vamos conversar sobre o `id` que será atribuído a cada objeto. Precisamos garantir que cada valor seja único, o que nos leva a pensar em um valor do tipo GUID, ou UUID. Caso você ainda não conheça este tipo de dado, recomendo uma leitura básica em https://pt.wikipedia.org/wiki/Identificador_%C3%BAnico_universal, que você pode fazer quando julgar interessante. Mas, em resumo,

tipos de dados GUID garantem sempre que seu valor NUNCA se repetirá.

Existe um componente no NPM que agrega a nossas aplicações a possibilidade de termos este tipo de dado disponível, o `guid-typescript`. Precisamos instalá-lo e faremos isso acessando o terminal na pasta de nossa aplicação. Estando lá, digite a seguinte instrução no prompt. Atente à versão que estamos utilizando.

```
npm i guid-typescript@1.0.9
```

Com nosso componente instalado, na pasta `src/app`, crie uma pasta chamada `models` e, dentro dela, o arquivo `peca.model.ts`, tal qual vemos no código a seguir. Observe a importação do componente instalado.

```
import { Guid } from 'guid-typescript';

export interface Peca {
    id: Guid;
    nome: string;
    valor: number;
}
```

Vamos criar uma pasta chamada `pages` dentro de `src/app` e, dentro dela, uma chamada `pecas` e, por fim, nesta pasta, outra, chamada `add-edit`. Nesta, vamos criar três arquivos: `pecas-add-edit.page.html`, `pecas-add-edit.page.ts` e `pecas-add-edit.module.ts`. Vamos aos códigos.

pecas-add-edit.page.html

Observe no código a seguir o caminho para a figura que utilizamos e lembre-se de que a pasta `assets` está no mesmo nível de `app` e você precisará criar a `imgs` e copiar a figura para esta pasta. Já aproveitamos neste momento para deixar o template preparado para a edição e inserção de peças, tal qual fizemos para `Tipos de Serviços`.

```
<ion-header>
    <ion-toolbar>
        <ion-buttons slot="start">
            <ion-back-button defaultHref=""></ion-back-button>
        </ion-buttons>
        <ion-title>Peças</ion-title>
    </ion-toolbar>
</ion-header>

<ion-content padding>
    <ion-item-divider color="light">
        <ion-grid no-padding>
            <ion-row no-padding>
                <ion-col col-3 align-self-center>
                    <ion-img style="width: 50px; height: 50px;"
src="assets/imgs/tab_pecas.png"></ion-img>
                </ion-col>
                <ion-col col-9>
                    <ion-label style="font-size: 30px">Dados da peça</ion-
label>
                </ion-col>
            </ion-row>
        </ion-grid>
    </ion-item-divider>

    <form [formGroup]="pecasForm" (submit)="submit()">
        <ion-item lines="none" [disabled]="!modoDeEdicao">
            <ion-label position="floating" class="label-input-
title">Nome</ion-label>
            <ion-input formControlName="nome" type="text" style="margin-
left: 3%;"></ion-input>
        </ion-item>

        <ion-item lines="none" [disabled]="!modoDeEdicao">
            <ion-label position="floating" class="label-input-
title">Valor</ion-label>
            <ion-input text-right formControlName="valor" type="number">
</ion-input>
        </ion-item>

        <ion-button *ngIf="!modoDeEdicao" shape="round" color="primary"
```

```
expand="block" padding (click)="iniciarEdicao()">
            Alterar dados
        </ion-button>
        <div *ngIf="modoDeEdicao" no-padding>
            <ion-grid>
                <ion-row>
                    <ion-col col-6 >
                        <ion-button shape="round" color="success"
size="small" padding type="submit" expand="block">
                            Gravar
                        </ion-button>
                    </ion-col>

                    <ion-col col-6>
                        <ion-button shape="round" color="warning"
size="small" padding (click)="cancelarEdicao()"  expand="block">
                            Cancelar
                        </ion-button>
                    </ion-col>
                </ion-row>
            </ion-grid>
        </div>
    </form>
</ion-content>
```

pecas-add-edit.page.ts

Observe que já temos a implementação praticamente concluída, restando-nos apenas pontos específicos deste capítulo, relacionados à persistência, e que veremos de maneira dedicada logo na sequência. Veja que também fazemos uso de nosso `Toast Service`, que já temos implementado no capítulo anterior e podemos reutilizá-lo. Temos alguns comentários do que precisamos ainda implementar. Verifique o uso do componente `Guid` no método `OnInit()`.

```
import { Component, OnInit } from '@angular/core';
import { Peca } from 'src/app/models/peca.model';
import { FormGroup, FormBuilder, Validators } from '@angular/forms';
import { ActivatedRoute } from '@angular/router';
import { Guid } from 'guid-typescript';
import { ToastService } from 'src/app/services/toast.service';

@Component({
    templateUrl: './pecas-add-edit.page.html'
})
export class PecasAddEditPage implements OnInit {
    private peca: Peca;
    public modoDeEdicao = false;
    public pecasForm: FormGroup;

    constructor(
        private formBuilder: FormBuilder,
        private route: ActivatedRoute,
        private toastService: ToastService,
    ) {}

    iniciarEdicao() {
        this.modoDeEdicao = true;
    }

    cancelarEdicao() {
        this.pecasForm.setValue(this.peca);
        this.modoDeEdicao = false;
    }

    submit() {
        // Atualizar/Inserir objeto de maneira persistida
        this.toastService.presentToast('Gravação bem sucedida', 3000,
'top');
        this.modoDeEdicao = false;
        // Navegar para a página principal
    }

    ngOnInit() {
        const id = this.route.snapshot.paramMap.get('id');
```

```
        if (id && Guid.isGuid(id)) {
            // Recuperaremos o objeto persistido
        } else {
            this.peca = {id: Guid.createEmpty(), nome: '', valor: 0.00 };
            this.modoDeEdicao = true;
        }

        this.pecasForm = this.formBuilder.group({
            id: [this.peca.id],
            nome: [this.peca.nome, Validators.required],
            valor: [this.peca.valor, Validators.required]
        });
    }

}
```

pecas-add-edit.module.ts

Para finalizarmos esta etapa e podermos testar a parte visual de nosso projeto, precisamos implementar o código para nosso módulo, e ele está na sequência.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { Routes, RouterModule } from '@angular/router';
import { IonicModule } from '@ionic/angular';
import { PecasAddEditPage } from './pecas-add-edit.page';

const routes: Routes = [
  {
    path: '',
    component: PecasAddEditPage
  }
];

@NgModule({
  imports: [
    CommonModule,
    FormsModule,
```

```
    IonicModule,
    RouterModule.forChild(routes),
    ReactiveFormsModule
  ],
  declarations: [PecasAddEditPage]
})
export class PecasAddEditPageModule {}
```

Precisamos alterar a rota inicial de nossa aplicação para a página que acabamos de implementar e devemos fazer isso no arquivo `app-routing.module.ts`, informando, na rota `home`, o caminho de nossa página criada, que é `'./pages/pecas/add-edit/pecas-add-edit.module#PecasAddEditPageModule'`.

Com esses códigos implementados, já podemos testar nossa aplicação. Veja a figura a seguir com a exibição da página para inserção de uma nova peça.

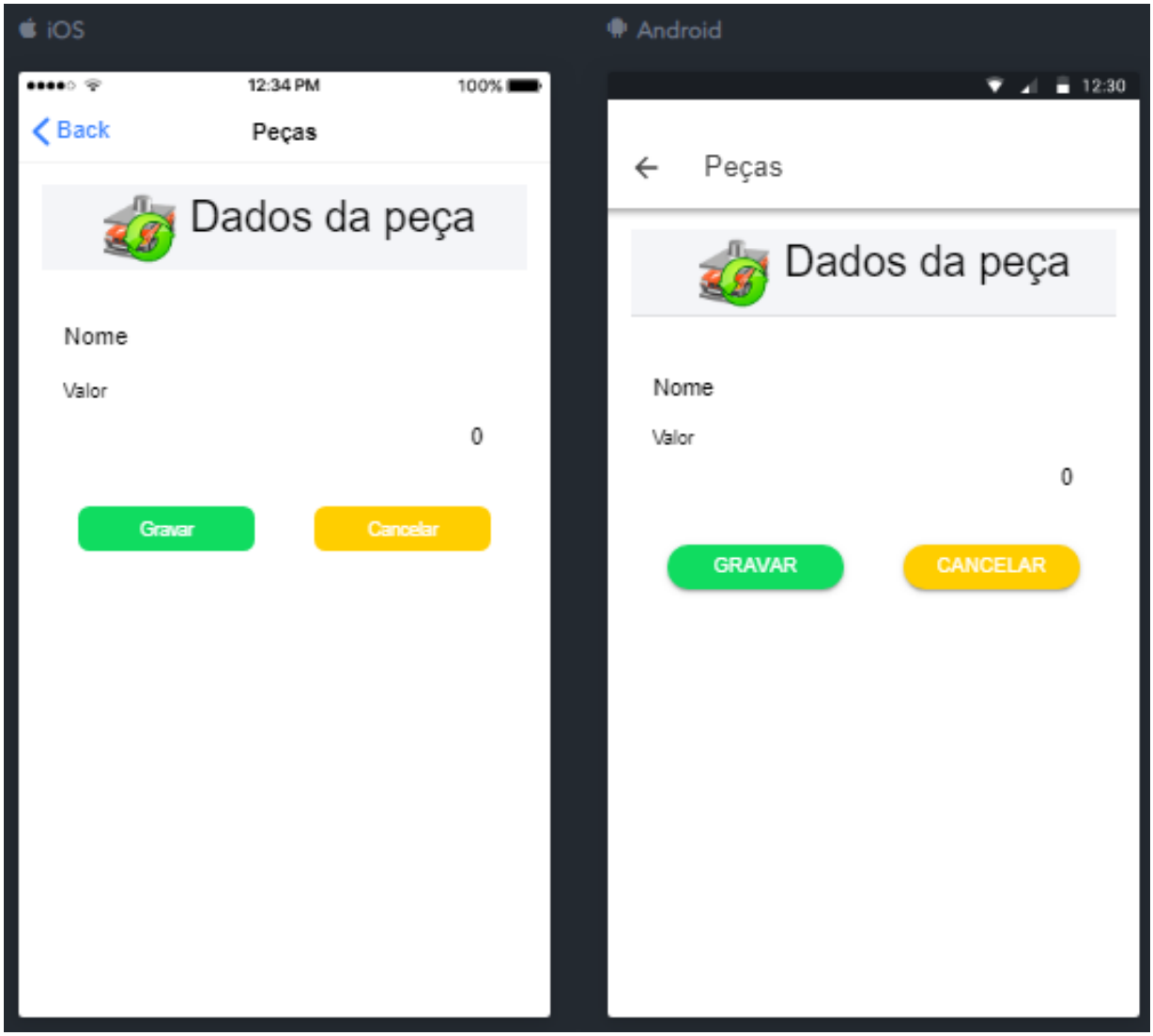


Figura 5.1: Página de inserção e alteração de peças

5.2 Persistência no Ionic com Storage para a inserção de dados

O Ionic possui um mecanismo de persistência nativo, ligado a uma API chamada `Storage`, que faz uso de grande variedade de mecanismos de persistência, dentre os quais você pode escolher o

melhor, em sua análise, para as plataformas que estiver desenvolvendo.

Inicialmente não especificaremos nenhum mecanismo de persistência, deixaremos que o Ionic utilize seu mecanismo nativo, que é o `IndexedDB`, disponibilizado no lado cliente, que em nosso caso é a engine do navegador.

Para implementarmos nosso primeiro código relacionado à inserção de uma peça, em uma base de dados do `IndexedDB`, precisamos instalar um plugin em nosso projeto, e isso pode ser feito no terminal, na pasta dele, com a instrução a seguir.

```
npm install @ionic/storage@2.2.0
```

Uma vez que o componente esteja instalado em nossa aplicação, precisamos disponibilizá-lo para nosso componente e, para isso, inseriremos as seguintes instruções.

```
// Junto aos demais imports de app.module.ts

import { IonicStorageModule } from '@ionic/storage';

// Ao final dos imports dentro do NgModule de app.module.ts
, IonicStorageModule.forRoot()
```

Agora que temos nosso mecanismo para persistência disponibilizado para nossa aplicação, precisamos implementar um serviço que atenderá às interações que exijam busca e armazenamento de dados na base. Na pasta `services`, crie um arquivo chamado `pecas.service.ts`, com o código a seguir.

```
import { Injectable } from '@angular/core';
import { Peca } from '../models/peca.model';
import { Guid } from 'guid-typescript';
import { Storage } from '@ionic/storage';

@Injectable({
  providedIn: 'root'
})
```

```
export class PecasService {

  constructor(
    private storage: Storage
  ) { }

  update(peca: Peca) {
    if (peca.id.isEmpty()) {
      peca.id = Guid.create();
    }

    this.storage.set(peca.id.toString(), JSON.stringify(peca));
  }
}
```

Veja que importamos `Storage` e o recebemos por injeção pelo construtor. Temos definido, inicialmente, o método para atualizar a base de dados. Dê uma lida na última instrução, que invoca o `set()` do objeto `storage`. Consegue identificar que estamos armazenando um par `chave/valor`? Pois bem, é assim que o `IndexedDB` trabalha, e a chave precisa ser uma string. A lógica do método `update()` é simples: caso seja um novo objeto, sem `id`, este valor será então gerado. Após essa verificação, o objeto é atribuído à nossa base.

Precisamos injetar nosso serviço no construtor. Sei que já fizemos isso, mas para reforçar, veja as instruções a seguir.

```
// Junto com demais imports do componente pecas-add-edit.page.ts
import { PecasService } from 'src/app/services/pecas.service';

// Código a ser inserido no construtor
private pecasService: PecasService
```

Vamos fechar esta etapa digitando as instruções a seguir logo no início do método `submit()`. Estamos invocando o `update()` e enviando o formulário completo para atualização e, na segunda instrução, atribuindo a `this.peca` o novo objeto, agora gravado.

```
this.pecasService.update(this.pecasForm.value);
this.peca = this.pecasForm.value;
```

Teste sua aplicação no navegador, informe o nome da peça e seu respectivo valor. Clique no botão gravar. Vamos ver como os dados foram persistidos?

Utilizando o Chrome, acesse as ferramentas do desenvolvedor (F12) e, em `Application`, localize a base de dados e veja no painel os dados que foram armazenados. Consegue se localizar na figura a seguir? O que achou?

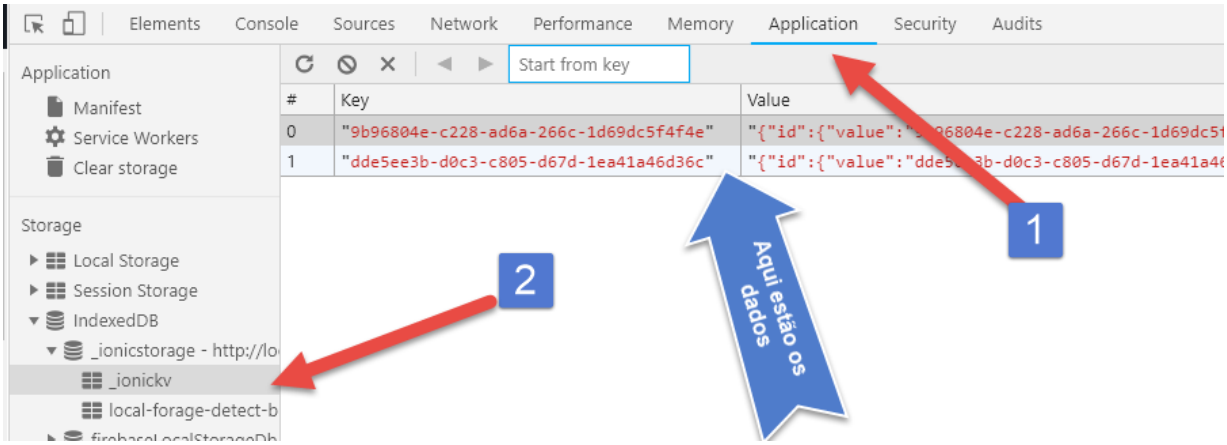


Figura 5.2: Verificação dos dados persistidos no IndexedDB

Caso seus dados não apareçam como dito anteriormente, procure no `WebSQL`.

5.3 Recuperação de dados persistidos com Storage

No capítulo anterior, começamos nossa aplicação com a página de listagem de dados e, a partir dela, navegávamos para uma operação de consulta, alteração, remoção ou inclusão. Neste capítulo, a escolha foi começar com a inserção, pois, como os dados são persistidos em uma base de dados, precisamos deles para implementar as demais funcionalidades.

Como já temos os dados persistidos, podemos começar a trabalhar com a página de listagem dos elementos armazenados e faremos isso com a implementação de um método em nossa classe de serviços para peças, a `pecas.service.ts`. Veja este código na sequência. Observe a primeira instrução do método, que parece um comentário, mas é uma diretiva para o TypeScript ignorar a indicação por preferência ao uso de constantes.

```
async getAll() {
    // tslint:disable-next-line:prefer-const
    let pecas: Peca[] = [];
    try {
      await this.storage.forEach((value: string, key: string) => {
        const peca: Peca = JSON.parse(value);
        pecas.push(peca);
      });
      return pecas;
    } catch (error) {
      return error;
    }
}
```

No código anterior, declaramos um array, que será populado com os objetos recuperados da base de dados por meio da invocação assíncrona ao método `forEach()`. Uma vez que todos os objetos estejam recuperados e não haja erro durante o processo de recuperação, a matriz é retornada ao chamador.

Resta-nos implementar nossa página de listagem de peças. Faremos isso criando dentro da pasta `pages/pecas` uma nova, chamada `listagem`, e nela os três arquivos necessários para nossa página, tal qual segue nas subseções.

pecas-listagem.page.html

Se verificar no `routerLink` para o primeiro `<ion-item>`, notará que o binding está ligado a um método, o `idAsString()`. Precisamos deste artifício para que o parâmetro possa ser enviado corretamente, como `String`, uma vez que o temos armazenado como `Guid`. Isso poderia ser contornado se criássemos uma propriedade do tipo `String` em vez de `Guid` e, na hora de armazenarmos o valor nela, o convertêssemos. Ainda, poderíamos utilizar a definição do modelo em uma classe e não interface e, então, definir na classe um método que resolvesse a conversão na obtenção do dado. Aqui optei por essa técnica para que possamos ver didaticamente a conversão de um dado no registro de links para rotas.

```
<ion-header>
    <ion-toolbar>
      <ion-title>Peças</ion-title>
    </ion-toolbar>
</ion-header>

<ion-content padding-left padding-top>
  <ion-list #slidingList>
    <ion-item-sliding *ngFor="let peca of pecas">
      <ion-item routerLink='/add-edit/{{idAsString(peca.id)}}'>
        <ion-label>
          <h2>{{peca.nome}}</h2>
          <p style="text-align:right">R$ {{peca.valor}}</p>
        </ion-label>
      </ion-item>
    </ion-item-sliding>
  </ion-list>

  <ion-fab vertical="top" horizontal="end" slot="fixed" edge="true">
    <ion-fab-button>
      <ion-icon name="add"></ion-icon>
    </ion-fab-button>
  </ion-fab>
</ion-content>
```

pecas-listagem.page.ts

Nosso `ngOnInit()` traz o uso de uma `Promise` para popular nossa coleção que será utilizada pelo template. Isso é necessário pelo fato de termos o `getAll()` definido como assíncrono. Poderíamos optar por transformar o `ngOnInit()` em assíncrono e realizar a invocação com `await`, em vez de trabalhar o `Promise`. Note também a implementação do método `idAsString()` para converter o `id` de `Guid` para `String` e em seguida para formato JSON.

Um detalhe: a documentação da biblioteca `Guid` aponta que, para retornar uma string de um valor Guid, bastaria invocar o método `toString()` da instância desejada, mas isso não funcionou aqui, mas funcionou na inserção, lembra? Na realidade, quando `id` chega, ao ser verificado, vem como `object Object`, o que impediu certamente a execução deste processo, por isso o *workaround* utilizado no método.

```
import { Component, OnInit, ViewChild } from '@angular/core';
import { PecasService } from '../../../services/pecas.service';
import { Peca } from '../../../models/peca.model';
import { ToastService } from '../../../services/toast.service';
import { IonList } from '@ionic/angular';
import { Guid } from 'guid-typescript';

@Component({
    templateUrl: './pecas-listagem.page.html'
})
export class PecasListagemPage implements OnInit {

    public pecas;
    @ViewChild('slidingList') slidingList: IonList;

    constructor(
        private pecasService: PecasService,
        private toastService: ToastService) {
    }

    idAsString(id: Guid): string {
        const convertedId = JSON.parse(JSON.stringify(id));
        return convertedId.value;
```

```
    }

    ngOnInit(): void {
        this.pecasService.getAll().then(pecas => {
            this.pecas = pecas;
        });
    }
}
```

pecas-listagem.module.ts

Finalizando a implementação dos três arquivos, temos o código a seguir para o arquivo nomeado no título da seção.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { IonicModule } from '@ionic/angular';
import { RouterModule, Routes } from '@angular/router';
import { PecasListagemPage } from './pecas-listagem.page';

const routes: Routes = [
    { path: '', component: PecasListagemPage }
  ];

@NgModule({
    imports: [
      CommonModule,
      FormsModule,
      IonicModule,
      RouterModule.forChild(routes)
    ],
    declarations: [
        PecasListagemPage
    ]
})

export class PecasListagemPageModule {}
```

Com as implementações anteriores realizadas, precisamos alterar nossa rota `home` e criar uma específica para a página de inserção e

alteração. Veja no código a seguir essas implementações, que devem ser realizadas no `app-routing.module.ts` .

```
const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', loadChildren: './pages/pecas/listagem/pecas-
listagem.module#PecasListagemPageModule' },
  { path: 'add-edit/:id', loadChildren: './pages/pecas/add-edit/pecas-add-
edit.module#PecasAddEditPageModule' }
];
```

A figura a seguir a traz em execução em nossos pseudoemuladores.

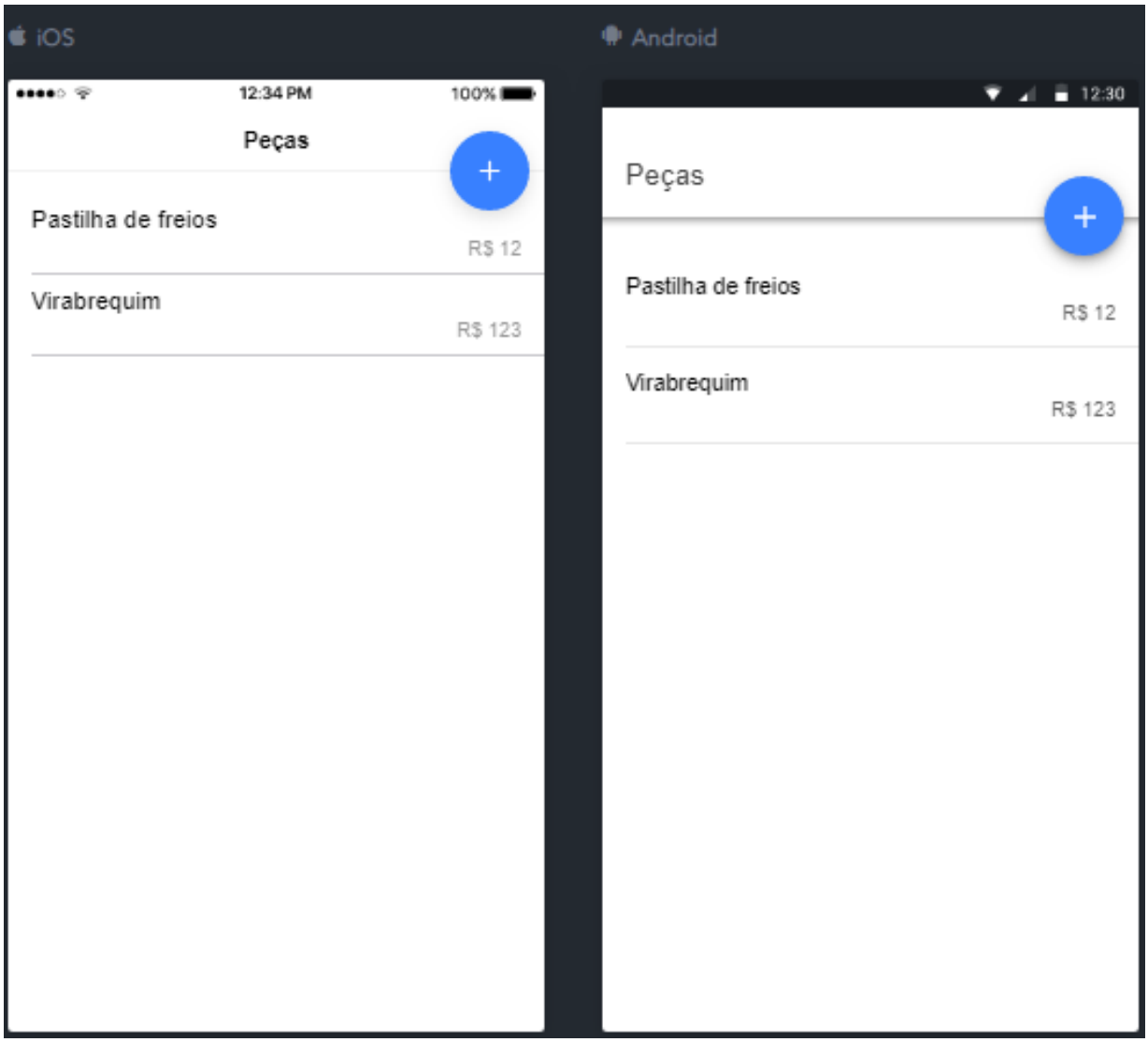


Figura 5.3: Listagem dos dados persistidos no IndexedDB

5.4 Recuperação de uma peça selecionada na listagem

No capítulo anterior, ao pressionarmos um item na página de listagem, éramos direcionados para a página de exibição e alteração dos dados. Precisamos realizar este procedimento aqui também. Teremos que recuperar os dados de nossa base e não mais de um

array. Vamos implementar o método em nossa classe de serviços para peças, a `pecas.service.ts`, assim:

```
async getById(id: string): Promise<Peca> {
    const pecaString = await this.storage.get(id);
    return JSON.parse(pecaString);
}
```

Veja que optamos por definir o tipo de dado de retorno aqui. Quando implementamos o `getAll()`, optamos por não definir de maneira explícita. Como instruções do método, temos a recuperação do valor na base, pelo `id` recebido, e então, com a peça já recuperada, a transformamos de `String` para um objeto JSON.

Vamos mudar um pouco nosso método `ngOnInit` de nosso arquivo `pecas-add-edit.page.ts` para podermos receber o `id` que virá da listagem quando um item for selecionado. O primeiro passo é informarmos que ele será assíncrono, inserindo `async` antes do nome do método. Depois, no `if()` que verifica se recebemos um `id` e se ele é um `Guid`, substituímos o comentário que está lá pelo código a seguir.

```
this.peca = await this.pecasService.getById(id);
```

Com isso temos as alterações necessárias quase finalizadas. Se formos testar nossa aplicação agora, veremos que, ao selecionarmos uma peça, a navegação ocorre, mas não vemos os dados da peça na página de alteração. Isso ocorre por um erro na criação do formulário de validadores (nosso `FormGroup`), que será invocada antes de termos recuperado a peça da base. Precisamos dizer ao nosso template que renderize nosso formulário apenas quando nosso `FormGroup` estiver devidamente criado. Para isso, envolvemos nossas tags `<form></form>` com as tags a seguir.

```
<ng-container *ngIf="pecasForm">
</ng-container>
```

Agora sim. Veja-a em execução na figura a seguir, que traz os dados de uma peça selecionada na página de listagem.

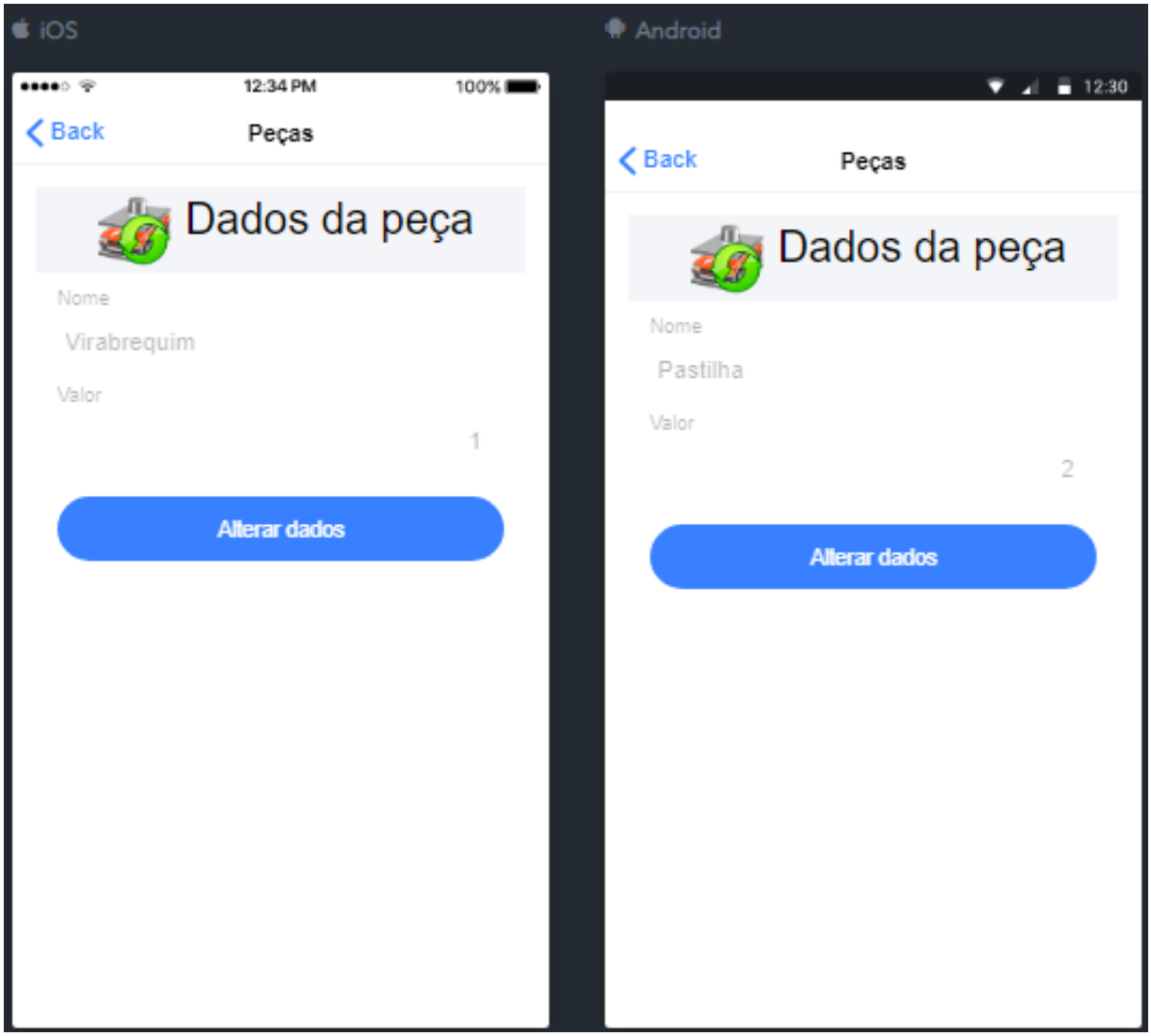


Figura 5.4: Visualização de um dado recuperado da base

5.5 Implementação e validação das operações CRUD

Vamos trabalhar na finalização do CRUD com a adoção da página de listagem dos dados. Já temos a visualização, vamos às demais.

Alteração nos dados de uma peça

Se você, ao executar a aplicação na última solicitação, clicou no botão para permitir a alteração dos dados e depois em gravar, viu que nada ocorreu, pois há um erro de execução, que é exibido na console do navegador e os dados não são alterados. O erro ocorre na invocação do método `isEmpty()` da propriedade `id`, em nossa classe `PecasService`, no método `update()`.

Nosso `submit()` envia os dados do formulário para `update()` e, na construção do formulário, a propriedade `id` da peça em vista perde a referência para o tipo `Guid`, desconhecendo o método `isEmpty()`. Vamos realizar um *workaround* para essa situação. Vamos inserir a instrução a seguir como início do método `update()`. Depois, teste novamente sua aplicação.

```
peca.id = Guid.parse(JSON.parse(JSON.stringify(peca.id)).value);
```

Remoção de uma peça da base de dados

Nossa implementação aqui refere-se à remoção de um item da base e você verá que isso é bem simples. Precisamos inserir a exibição da opção de remover para o usuário. Faremos isso pelo código a seguir, no template, antes de fechar o `<ion-item-sliding>`.

```
<ion-item-options side="end">
    <ion-item-option color="danger" (click)="removerPeca(peca)">
      <ion-button color="danger" no-padding>
          <ion-icon name="trash"></ion-icon>
          Remover
      </ion-button>
    </ion-item-option>
</ion-item-options>
```

Temos que implementar um método em `pecas.service.ts` e outro em `pecas-listagem.page.ts` e eles estão na sequência.

```
// pecas.service.ts
async removeById(id: string) {
```

```
    await this.storage.remove(id);
}

// pecas-listagem.page.ts
async removerPeca(peca: Peca) {
    await this.pecasService.removeById(this.idAsString(peca.id));
    this.pecas = await this.pecasService.getAll();
    this.toastService.presentToast('Peça removida', 3000, 'top');
    await this.slidingList.closeSlidingItems();
}
```

No segundo método, observe que enviamos para o primeiro uma string que representa o `id` da peça selecionada. Após a remoção, repopulamos nosso objeto utilizado na renderização da página. Terminamos com a exibição de mensagem e podemos testar a funcionalidade.

Inserção por meio do FAB Button

No início do capítulo realizamos o processo de inserção por meio do acesso direto à página com esta funcionalidade. Precisamos adaptar nosso template para que essa invocação possa ocorrer. Veja a nova implementação para o componente na sequência.

```
<ion-fab-button routerLink='/add-edit/-1'>
```

Vamos adaptar nosso método `submit` para promover a navegação para nossa página de listagem após nossa gravação. Da maneira como estamos fazendo, tanto faz a operação que o usuário esteja realizando, alteração ou inclusão, ao final, ele será redirecionado para a listagem. Veja a seguir o código, que deve ser implementado como última instrução do `submit()`, em `pecas-add-edit.page.ts`, antecedido por uma instrução a ser inserida no construtor.

```
// Inserir no construtor. Atenção ao import
private router: Router

// Última instrução do submit()
this.router.navigateByUrl('');
```

O que acha de agora testarmos todas as funcionalidades de nossa aplicação? Notou que a peça inserida não aparece em nossa listagem? Pois é. Isso está relacionado ao ciclo de vida de uma visão, nossa página.

Temos em `pecas-listagem.page.ts` a implementação do método `ngOnInit()`, que é disparado quando a página é inicializada da primeira vez. É neste método que temos a população de nosso objeto utilizado para renderização da listagem.

Como dito, este evento ocorre só na primeira vez que a página é criada. Quando navegamos da inserção ou alteração para a listagem, a página já existe. Precisamos implementar a lógica do `ngOnInit()` em outro método, de outro evento. Você pode optar agora por apagar o comportamento deste método, deixando-o vazio, e então implementar o método da listagem a seguir.

```
ionViewWillEnter() {
    this.pecasService.getAll().then(pecas => {
        this.pecas = pecas;
    });
}
```

Veja na listagem anterior o prefixo `ion`, que deixa claro que este método captura um evento do ciclo de vida de uma página/visão, mas agora estamos trabalhando eventos do Ionic e não do Angular. Vamos testar novamente nossa aplicação e ver se o problema foi corrigido?

5.6 SQLite e Cordova como mecanismo de persistência

Na seção anterior, realizamos operações de CRUD tendo como base o Storage do Ionic. Vimos que os dados ficam atrelados ao navegador e que o acesso a eles se dá por meio de conjunto de

chave/valor. Não fizemos nada grande, mas é importante saber que este mecanismo não é o ideal para uma persistência de dados ao estilo de uma base de dados.

Felizmente temos alguns bancos de dados que funcionam bem em dispositivos móveis, com características de tabelas e consultas. Dentre eles, o mais popular e bem aceito é o SQLite (https://www.sqlite.org/index.html). Nós o utilizaremos em nossa aplicação por meio de um plugin do Cordova, executando as instruções a seguir.

```
// Caso o Cordova não esteja instalado em seu ambiente, precisamos
primeiro fazer isso
npm i -g cordova

// Instala a integração de seu projeto com o Cordova, mas se já fez
anteriormente, não precisa novamente.
ionic integrations enable cordova --add

// Atualize o arquivo config.xml como orientado no Capítulo 2

// Instalação do plugin para o SQLite. Note a versão utilizada
ionic cordova plugin add cordova-sqlite-storage
npm install @ionic-native/sqlite@5.7.0

// Adicionar as plataformas Android e iOS ao projeto. Pode optar no
momento apenas pela que quiser
ionic cordova platform add android@8.0.0
ionic cordova platform add ios@5.01
```

Precisaremos realizar algumas implementações relacionadas à configuração da aplicação em nosso arquivo `app.module.ts` e, conforme elas forem necessárias, vamos realizando. Para iniciar, precisamos importar o plugin do SQLite e fazemos isso com o código a seguir.

```
import { SQLite } from '@ionic-native/sqlite/ngx';
```

Na sequência vamos inserir o seguinte código em nosso objeto `providers`, dentro do `@NgModule`, após os já existentes. Esse código

subsidia a formatação de datas e números em português do Brasil e traz para a aplicação o plugin do SQLite. Lembre-se de inserir o `import` com o assistente do VSC, para o `LOCALE_ID`.

```
{ provide: LOCALE_ID, useValue: 'pt-BR' },
SQLite
```

Para garantir que a localização e internacionalização funcionem corretamente em toda a aplicação, no mesmo arquivo, antes de `@NgModule()`, insira o seguinte código.

```
import { registerLocaleData } from '@angular/common';
import localePt from '@angular/common/locales/pt';

registerLocaleData(localePt, 'pt-BR');
```

Vamos agora partir para a implementação do serviço que nos proverá recursos relacionados à base de dados SQLite que utilizaremos em nossa aplicação. Para isso, na pasta `src/app/services`, vamos criar um arquivo chamado `database.service.ts`, com o código inicial igual ao apresentado na sequência. Verifique que o construtor recebe por injeção um objeto para o nosso plugin.

```
import { Injectable } from '@angular/core';
import { SQLite } from '@ionic-native/sqlite/ngx';

@Injectable({
    providedIn: 'root'
})
export class DatabaseService {
    constructor(
        private sqlite: SQLite
    ) { }
}
```

Nosso primeiro método a ser implementado será o responsável por retornar nossa base de dados. Embora estejamos invocando o método `create()` de nosso plugin, caso a base já exista, a criação

não ocorre e o método apenas retorna a já existente. Veja o código a seguir.

```
public getDatabase() {
    return this.sqlite.create({
      name: 'oficina.db',
      location: 'default'
    });
}
```

Com o método responsável pela criação de nossa base de dados pronto, precisamos implementar a criação das tabelas e nosso foco nesta seção será a tabela responsável pelo registro da entrada do veículo na oficina, ou seja, a criação de uma ordem e serviço. Veja o código para este método na sequência. Observe que já prevemos o relacionamento com um cliente, por meio do campo `clienteid`. Veja também que o método é assíncrono, pois o `sqlBatch()` retorna uma `Promise`, que também é o tipo de dado retornado por nosso método. Não entrarei em detalhes sobre as instruções para criação de tabelas, pois SQL não é nosso objetivo. Uma observação muito importante quando trabalhamos com `Promises` ou métodos assíncronos é tipificarmos seu retorno, não deixando o `any`, pois isso pode fazer com que chamadas assíncronas ocorram de maneira paralela, causando um resultado indesejável. O último detalhe a ressaltar deste método é seu escopo, privado.

```
private async createTables(db: SQLiteObject): Promise<boolean> {
    try {
        await db.sqlBatch([
            ['CREATE TABLE IF NOT EXISTS ordensdeservico (ordemdeservicoid
TEXT primary key NOT NULL, ' +
                'clienteid TEXT NOT NULL, veiculo TEXT NOT NULL,
dataehoraentrada DATETIME NOT NULL, ' +
                'dataehoratermino DATETIME, dataehoraentrega DATETIME)']
            ])
        .then( () => {
            console.log('Criação de tabelas realizada com sucesso');
            return true;
        });
```

```
    } catch (e) {
        console.error('Erro na criação das tabelas', e);
        return false;
    }
}
```

Até aqui temos implementadas a criação da base de dados e de nossa primeira tabela no SQLite e, seguindo nosso último exemplo de testes, começaremos com a listagem de ordens de serviço, mas ainda não temos nenhuma. Podemos pensar em implementar a página de adição e alteração, como já temos para os demais modelos de negócio, mas vamos pensar em popular nossa base com alguns dados para testes. Veja então o método a seguir, que devemos implementar no arquivo `database.service.ts` que estávamos trabalhando.

```
private async populateDatabase(db: SQLiteObject): Promise<boolean> {
    try {
        await db.executeSql('select COUNT(ordemdeservicoid) as qtdeOS from
ordensdeservico', [])
            .then ( async (data) => {
                if (data.rows.item(0).qtdeOS === 0) {
                    try {
                        const clienteID = Guid.create().toString();
                        const ordemDeServicoID = Guid.create().toString();
                        await db.sqlBatch([
                            ['insert into ordensdeservico ' +
                                '(ordemdeservicoid, clienteid, veiculo,
dataehoraentrada) values (?, ?, ?, CURRENT_TIMESTAMP)',
                                [ordemDeServicoID, clienteID, 'ABC-1235']]
                        ])
                        .then( () => {
                            console.log('Base de dados populada com
sucesso');
                            return true;
                        });
                    } catch (e) {
                        console.error('Falha ao popular base de dados',
e);
                        return false;
```

```
                    }
                }
            });
    } catch (e) {
        console.error('Erro ao consultar a quantidade de registros já
inseridos', e);
        return false;
    }
}
```

Observe que realizamos uma consulta que nos retornará a quantidade de registros que temos na tabela que estamos criando nesta seção. Caso não existam registros, inserimos então os iniciais. Novamente reforço que não entrarei em detalhes sobre SQL. Mas quando este método será invocado?

Quando inicializarmos nossa aplicação, podemos ter a necessidade de invocar um método que possa encapsular a realização de diversas tarefas em nossa base de dados, em vez de, por exemplo, transformarmos o método anterior em público e invocá-lo. Pensando nisso, vamos implementar o método que está na sequência, também assíncrono. Veja que invocamos os métodos anteriores, enviando nossa base de dados a eles.

```
public async createDatabase(): Promise<boolean> {
    try {
        await this.getDatabase()
            .then( async (db: SQLiteObject) => {
                await this.createTables(db)
                    .then( async () => await this.populateDatabase(db) )
                    .then( () => true );

            });
    } catch (e) {
        console.error('Erro na criação da base de dados', e);
        return false;
    }
}
```

Agora, para que nossa aplicação possa garantir que a base de dados seja criada e esteja disponível para nossa aplicação, precisamos injetar este serviço implementado em nosso arquivo `app.component.ts` e então invocar o método anterior. A injeção, no construtor, se dá pela inserção da instrução a seguir.

```
private databaseService: DatabaseService
```

Na sequência, precisamos adaptar o método `initializeApp()`. As alterações em relação ao código que você deve ter já implementado estão em transformar a função anônima em assíncrona (async) e na invocação ao método `createDatabase()`.

```
async initializeApp() {
    this.platform.ready().then(async () => {
      this.statusBar.styleDefault();
      await this.databaseService.createDatabase()
        .then( () => this.splashScreen.hide());
    });
}
```

Poderíamos executar nossa aplicação agora, mas nada será exibido, pois não temos as páginas para Ordens de Serviço criadas. Vamos fazer isso em nossa próxima seção.

5.7 A interface com o usuário para as Ordens de Serviço

Para iniciarmos a etapa de criação de nossa interface com o usuário, estamos adotando a prática de criarmos inicialmente a página de listagem dos dados de nosso modelo de negócio. Desta maneira, teremos uma repetição de implementação em relação à técnica aqui adotada, mas a ideia é trazer o código para você, para facilitar o acompanhamento. Assim, as próximas subseções terão estes códigos.

A classe de modelo

Como estamos trabalhando sempre com uma interface que representa nosso modelo de negócio, vamos criar, na pasta `src/app/models`, um arquivo chamado `ordemdeservico.model.ts`, tal qual o código a seguir. Observe que inicialmente não temos como propriedade todos os campos que criamos em nossa tabela no SQLite.

```
export interface OrdemDeServico {
    ordemdeservicoid: string;
    clienteid: string;
    veiculo: string;
    dataehoraentrada: Date;
}
```

O Service para ordens de serviço

Na pasta `src/app/services`, vamos criar um arquivo chamado `ordensdeservico.service.ts`, tal qual apresentamos na sequência.

```
import { Injectable } from '@angular/core';
import { DatabaseService } from './database.service';
import { OrdemDeServico } from '../models/ordemdeservico.model';

@Injectable({
    providedIn: 'root'
})
export class OrdensDeServicoService {
  constructor(
      private databaseService: DatabaseService
  ) { }

  public async getAll() {
    try {
      const db = await this.databaseService.getDatabase();
      const sql = 'SELECT * FROM ordensdeservico';
      try {
        const data = await db.executeSql(sql, []);
        if (data.rows.length > 0) {
```

```
            // tslint:disable-next-line:prefer-const
            let ordensdeservico: OrdemDeServico[] = [];
            for (let i = 0; i < data.rows.length; i++) {
              const ordemdeservico = data.rows.item(i);
              ordensdeservico.push(ordemdeservico);
            }
            return ordensdeservico;
          } else {
            return [];
          }
        } catch (e) {
          return console.error(e);
        }
      } catch (e) {
        return console.error(e);
      }
    }
}
```

Verifique que injetamos um `DatabaseService` para a classe atual, que também é um `Service`. Implementamos um método básico para recuperação de todos os dados da tabela `ordensdeservico`, com todos os campos. A população das propriedades será automática, uma vez que o nome dos campos é o mesmo delas. Isso será realizado para cada registro recuperado.

NgModule para a página de listagem

Dentro de `src/app/pages`, vamos criar a pasta `ordensdeservico/listagem` e, dentro dela, nosso arquivo, chamado `ordensdeservico-listagem.module.ts`, tal qual temos apresentado na listagem a seguir.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { IonicModule } from '@ionic/angular';
import { RouterModule, Routes } from '@angular/router';
import { OrdensDeServicoListagemPage } from './ordensdeservico-
listagem.page';
```

```
const routes: Routes = [
    { path: '', component: OrdensDeServicoListagemPage }
  ];

@NgModule({
    imports: [
      CommonModule,
      FormsModule,
      IonicModule,
      RouterModule.forChild(routes)
    ],
    declarations: [
        OrdensDeServicoListagemPage
    ]
})

export class OrdensDeServicoListagemPageModule {}
```

Classe controladora para a página de listagem

Na mesma pasta em que criamos o arquivo anterior, vamos agora criar outro, chamado `ordensdeservico-listagem.page.ts` e vamos implementar nele o código a seguir. Temos a propriedade que será populada com os objetos recuperados, a injeção do Service e o método `ionViewWillEnter()` .

```
import { Component, OnInit, ViewChild } from '@angular/core';
import { OrdensDeServicoService } from
'src/app/services/ordensdeservico.service';
import { ToastService } from 'src/app/services/toast.service';
import { OrdemDeServico } from 'src/app/models/ordemdeservico.model';
import { IonList } from '@ionic/angular';

@Component({
    templateUrl: './ordensdeservico-listagem.page.html'
})
export class OrdensDeServicoListagemPage implements OnInit {

    public ordensDeServico: void | OrdemDeServico[] = [];
```

```
    @ViewChild('slidingList') slidingList: IonList;

    constructor(
        private ordensdeservicoService: OrdensDeServicoService,
        private toastService: ToastService
        ) { }

    ngOnInit() {

    }

    async ionViewWillEnter() {
        const oss = await this.ordensdeservicoService.getAll();
        this.ordensDeServico = oss;
    }
}
```

Template para a página de listagem

Novamente, na mesma pasta, vamos criar o arquivo `ordensdeservico-listagem.page.html` e inserir nele o código a seguir. Veja aqui que, com exceção dos dados do modelo de negócio, toda a implementação é semelhante ao que fizemos para `Peças`. Ainda não estamos incluindo comportamentos relacionados a seleção e inserção. Faremos isso na sequência.

```
<ion-header>
    <ion-toolbar>
      <ion-title>Ordens de serviço</ion-title>
    </ion-toolbar>
</ion-header>

<ion-content padding-left padding-top>
  <ion-list #slidingList>
    <ion-item-sliding *ngFor="let ordemDeServico of ordensDeServico">
      <ion-item>
        <ion-label>
          <h2>{{ordemDeServico.veiculo}}</h2>
          <p style="text-align:right">Entrada:
{{ordemDeServico.dataehoraentrada | date: 'dd/MM/yyyy'}}</p>
```

```
        </ion-label>
      </ion-item>

      <ion-item-options side="end">
        <ion-item-option color="danger">
          <ion-button color="danger" no-padding>
              <ion-icon name="trash"></ion-icon>
              Remover
          </ion-button>
        </ion-item-option>
      </ion-item-options>
    </ion-item-sliding>
  </ion-list>


  <ion-fab vertical="top" horizontal="end" slot="fixed" edge="true">
    <ion-fab-button>
      <ion-icon name="add"></ion-icon>
    </ion-fab-button>
  </ion-fab>
</ion-content>
```

Observe a instrução que exibirá a `dataehoraentrada`. Estamos utilizando um `Pipe` do Angular, que é herdado pelo Ionic. Ele permite a transformação de um dado para uma formatação específica, em nosso caso um `date` no formato `dd/MM/yyyy`.

5.8 A aplicação em execução

Se executarmos nossa aplicação agora, a página a ser exibida será a de peças e queremos a de ordem de serviço. Precisamos mudar `app-routing.module.ts`. Veja o código na sequência.

```
const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', loadChildren:
'./pages/ordensdeservico/listagem/ordensdeservico-
```

```
listagem.module#OrdensDeServicoListagemPageModule' },
  { path: 'pecas', loadChildren: './pages/pecas/listagem/pecas-
listagem.module#PecasListagemPageModule' },
  { path: 'add-edit/:id', loadChildren: './pages/pecas/add-edit/pecas-add-
edit.module#PecasAddEditPageModule' }
];
```

Não podemos testar esta funcionalidade nos pseudoemuladores. Desta maneira, você pode escolher conectar seu dispositivo em seu computador, ou executar um emulador. Sem a necessidade de executar o Android Studio, para testarmos a aplicação Android, execute a seguinte instrução no terminal.

```
ionic cordova run android -l
```

Com isso, será executado todo o processo necessário para compilação referente à criação da aplicação para o Android, inicialização do emulador, distribuição dela e execução. Você verá sua aplicação como mostra a figura a seguir. Aproveito para mostrá-la no emulador e na tela de console do Chrome, como também vimos no capítulo 2, o que nos possibilita depurar a aplicação. Veja as saídas na console que temos em nossas implementações.

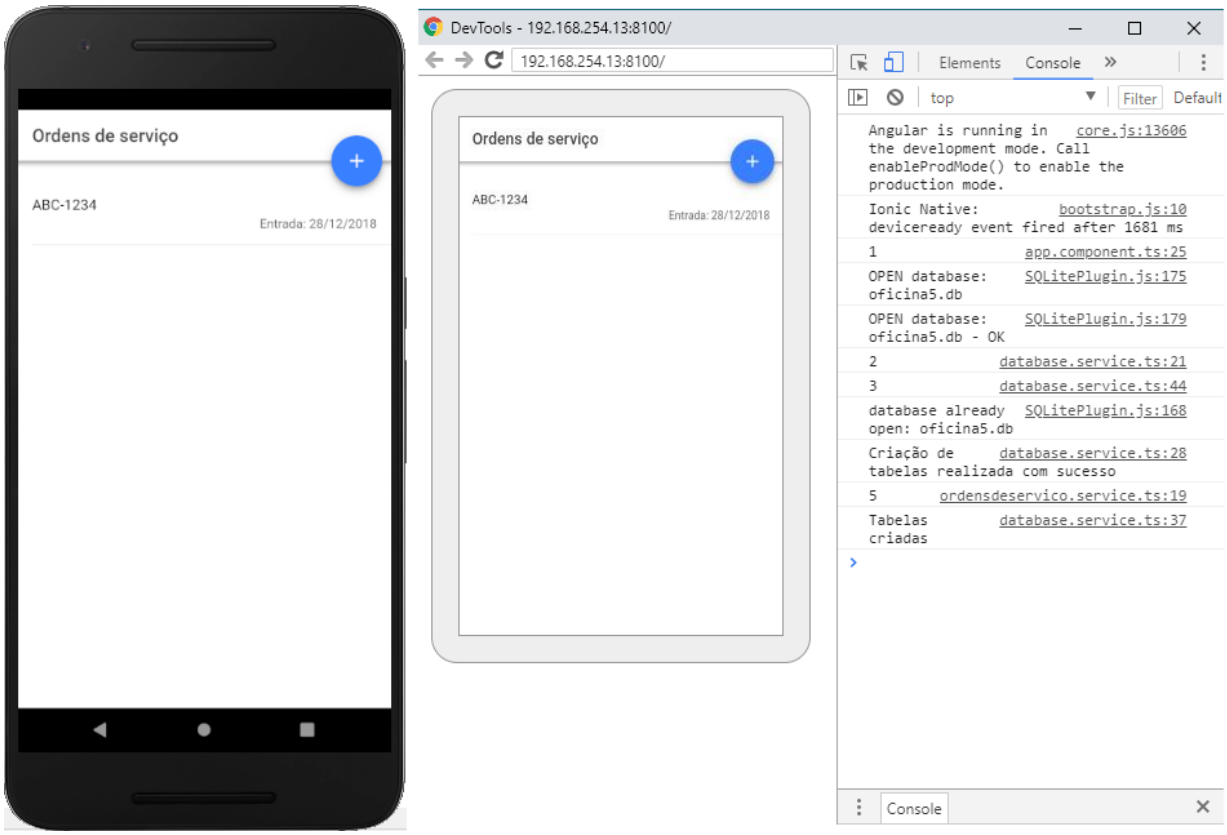


Figura 5.5: Aplicação Android em execução no emulador e console do Chrome

Vamos agora ao iOS? Vamos para a mesma estratégia que utilizamos há pouco para o Android, sem a necessidade de abrirmos o XCode. Se for utilizar um dispositivo no iOS, pode ser que seja necessário realizar toda a configuração de equipe no XCode na primeira execução, como visto no capítulo 2.

```
ionic cordova run ios -l
```

Com o sucesso na execução de uma das instruções anteriores, podemos ver nossa aplicação funcionando, como na figura a seguir.


12:25
Ordens de serviço
ABC-1234
Entrada: 29/12/2018

Figura 5.6: Aplicação iOS em execução no emulador do iPhone X

5.9 E o Capacitor?

Desde o capítulo 2 não vimos nada no livro dizendo diretamente para usar o Capacitor, deixando sempre a seu critério o que utilizar, entre ele e o Cordova. Vamos utilizar o Capacitor? Vamos às instruções a seguir.

```
ionic integrations enable capacitor

Configuração do capacitor.config.json

npm run build

ionic capacitor add android

ou

ionic capacitor add ios

Depois

ionic capacitor copy android

ou

ionic serve

Configurar o server no capacitor.config.json

Depois

ionic capacitor sync android

ou

ionic capacitor open android
```

Se você quiser se aprofundar no Capacitor, poderia começar com uma leitura no post https://blog.ionicframework.com/announcing-capacitor-1-0/.

Todos os plugins que utilizaremos neste livro são do Cordova, mas funcionam no Capacitor. Caso você foque apenas no Capacitor, pode ser interessante buscar por plugins desenvolvidos especificamente para ele.

Conclusão

Chegamos ao final deste interessante capítulo. Colocamos em prática tudo que vimos nos anteriores e adicionamos o recurso de persistência local, tanto por meio do Ionic Storage, que é nativo na plataforma, como pelo banco de dados SQLite, por meio de um plugin para o Cordova.

Criamos serviços para criação e obtenção do banco de dados e suas tabelas, assim como a inserção de dados que possam popular nossa base para testes. Reforçamos também o uso de emuladores para a execução das aplicações, agora diretamente pelo Ionic CLI e voltamos a utilizar o Capacitor.

Nossos testes com o SQLite se limitaram a uma listagem de registros existentes em uma tabela, que em nosso caso foi a de Ordens de Serviço. Não concluímos o CRUD neste capítulo, mas o faremos no próximo, onde abordaremos recursos que serão necessários para as funcionalidades faltantes.

Capítulo 6
Side Menu, CRUD com SQLite e o componente Select

Finalizamos o capítulo anterior com a criação e acesso a uma base de dados SQLite para manutenção e consulta em dados relacionados ao registro do atendimento de um veículo ao entrar em nossa oficina. No início do capítulo 5, vimos um CRUD completo com o Ionic Storage, mas não completamos o CRUD com o SQLite, pois isso ficaria muito extenso.

Agora, neste nosso sexto capítulo, daremos sequência e conclusão ao CRUD de Ordens de Serviço, conheceremos um estilo de menu para acesso às páginas da aplicação e veremos também um componente que tenha a funcionalidade de um ComboBox.

Se lembrarmos bem, nossa tabela de Ordem de Serviço possui uma associação com clientes, por meio de um campo chamado `ClienteID`, que representa a chave estrangeira para a tabela de clientes, mas nós também não criamos esta tabela no capítulo anterior. Trabalharemos tudo isso neste capítulo.

6.1 Criação do Sidemenu

É interessante que uma aplicação com diversas funcionalidades, representadas por páginas, possua um ponto de início, um menu, que permita o acesso a essas funcionalidades. Existem algumas variações para isso. Os menus em aplicações móveis têm adotado a característica conhecida como Sidemenu, que é uma página que não é exibida em sua totalidade e que pode ser deslizada para a esquerda (normalmente) para ser ocultada, ou então, puxada para a direita para ser exibida.

Vamos começar nossa implementação deste capítulo com a criação de nosso Sidemenu, que trará o acesso que precisamos para as funcionalidades de nossa aplicação. Como estamos organizando nossas páginas em pastas, dentro de `pages`, vamos então criar uma nova pasta, chamada `menu`. Dentro dela criaremos os arquivos que comporão esta funcionalidade e que veremos nas subseções seguintes.

O template para o menu

Dentro de nossa nova pasta (`pages/menu`), vamos criar um arquivo de CSS, com uma única classe de estilo, que utilizaremos no menu, para que a página em exibição apareça com um destaque ao lado esquerdo de sua exibição no menu. Vamos dar a este arquivo o nome `sidemenu.page.scss` e implementar o seguinte código nele.

```
.active-item {
    border-left: 8px solid var(--ion-color-primary);
}
```

Vamos ao arquivo responsável pelo HTML para nosso template. Daremos a ele o nome de `sidemenu.page.html` e implementaremos nele o código a seguir.

```
<ion-split-pane>
    <ion-menu contentId="content">
        <ion-header>
            <ion-toolbar>
                <ion-title>Meu Calhambeque - MENU</ion-title>
            </ion-toolbar>
        </ion-header>
        <ion-content>
            <ion-img style="width: 250px; height: 250px;"
src="assets/imgs/logo.png" padding></ion-img>
            <ion-list>
                <ion-menu-toggle *ngFor="let p of pages">
                    <ion-item [href]="p.url" [class.active-
item]="selectedPath === p.url">
                        <ion-img [src]="p.src" style="width: 32px; height:
```

```
32px;" ></ion-img>
                        <ion-label>
                            {{ p.title }}
                        </ion-label>
                    </ion-item>
                </ion-menu-toggle>
            </ion-list>
        </ion-content>
    </ion-menu>

    <ion-router-outlet name="menucontent" id="content" main></ion-router-
outlet>
</ion-split-pane>
```

O primeiro componente que temos em nosso template é o `<ion-split-pane>`. A documentação do Ionic traz a definição dele como um componente que torna possível a criação de um layout conhecido por `multi-view`, ou seja, a visualização de mais de uma página na mesma tela, muito comum em aplicações originárias no iPad, possibilitando o uso de menus.

Com o uso do `SplitPane`, se o tamanho da tela do dispositivo é menor que um determinado valor, o componente ocultará o menu, dando preferência à visão atualmente ativa. De acordo com a documentação, este componente é útil quando se desenvolve aplicações que serão executadas em dispositivos móveis, como tablets e telefones, ou para executar no navegador.

Dentro do `<ion-split-pane>` começamos a definir nosso menu, fazendo uso do componente `<ion-menu>`, que é o responsável pela criação dos componentes que representarão as opções ao usuário, assim como permitir que a página de menu seja deslizada, tanto para ocultá-lo, como para exibi-lo.

Observe que no primeiro `<ion-img>` definimos uma imagem que será utilizada como logomarca para a aplicação. Lembre-se de ter esta imagem fisicamente em seu projeto.

Dentro do componente `<ion-menu>`, temos o atributo `contentId`, recebendo o valor `content`, que será utilizado pelo componente responsável pela renderização do conteúdo das páginas acessadas pelas opções disponibilizadas pelo menu. Antecipando, este componente é o `<ion-router-outlet>` que está ao final do código. Já comentaremos sobre ele.

No componente `<ion-list>`, temos um novo, o `<ion-menu-toggle>`, que será responsável por gerenciar a abertura das páginas de conteúdo que a aplicação acessará a partir do menu. Ele garante a alternância entre e visível ou não para o menu. Temos no controle o `*ngFor`, que percorrerá uma coleção que definiremos na classe do template, que possibilitará a renderização dos itens do menu, por meio do conteúdo que temos no `<ion-item>`, onde temos um `<ion-img>` e um `<ion-label>` para cada item. Observe os bindings para estes componentes.

Ao final, temos o `<ion-router-outlet>`. Já o conhecemos lá do capítulo 2, onde vimos e estudamos a estrutura de uma aplicação Ionic. Observe que definimos o nome `menucontent` para ele, o id `content` que é o que utilizamos no `<ion-menu>` e temos o atributo `main`, que define o componente como principal da aplicação.

A classe TypeScript para nosso template

Com nosso HTML implementado, podemos começar a implementação para a classe que responderá pela renderização dele. Vamos então criar o arquivo para ela, na mesma pasta do template. Vamos nomeá-lo de `sidemenu.page.ts` e implementar nele o código a seguir.

```
import { Component, OnInit } from '@angular/core';
import { Router, RouterEvent } from '@angular/router';

@Component({
  selector: 'app-menu',
  templateUrl: './sidemenu.page.html',
  styleUrls: ['./sidemenu.page.scss'],
```

```
})
export class SideMenuPage implements OnInit {

    selectedPath = '';

    pages = [
        {
          title: 'Tipos de serviços',
          url: '/menu/(menucontent:tiposservicos)',
          src: 'assets/imgs/icon_tiposservicos.png'
        },
        {
          title: 'Peças',
          url: '/menu/(menucontent:pecas)',
          src: 'assets/imgs/tab_pecas.png'
        },
        {
          title: 'Clientes',
          url: '/menu/(menucontent:clientes)',
          src: 'assets/imgs/icon_clientes.png'
        },
        {
          title: 'Atendimentos',
          url: '/menu/(menucontent:atendimentos)',
          src: 'assets/imgs/icon_atendimentos.png'
        }
    ];

    constructor(private router: Router) {
        this.router.events.subscribe((event: RouterEvent) => {
          this.selectedPath = event.url;
        });
    }

    ngOnInit() {
    }

}
```

Logo no início do arquivo, temos a declaração de duas variáveis: `selectedPath` , e `pages` . A primeira será responsável por manter

sempre qual a página atualmente ativa. Isso nos ajudará a dar o destaque nela, por meio do CSS que criamos e do qual fizemos um binding no template com base nesta variável.

A segunda, é um array, com objetos que representam dados sobre as páginas, que também estamos utilizando no template, para a renderização das opções. Veja as imagens que precisaremos ter fisicamente em nossa aplicação. Observe a `url`, o conteúdo entre parênteses, `menucontent`, que é o nome que demos ao controle `<ion-router-outlet>`. Mais à frente tocaremos neste padrão.

O construtor da classe recebe por injeção um objeto `Router`, para que possa identificar qual a rota está sendo requisitada e então atualizar a variável `selectedPath`.

O roteamento a ser seguido pela aplicação

Nós já conhecemos o sistema de rotas que estamos sempre trabalhando em nossos testes, e normalmente trabalhamos isso no arquivo `app-routing.module.ts`. Agora, estamos definindo um menu para nossa aplicação e é por ele que as páginas serão acessadas. Isso nos leva à possibilidade de termos um arquivo de roteamento atrelado ao nosso módulo de menu e implementarmos nele a definição de todas as rotas a partir do menu, tornando-o o ponto de partida para as páginas de nossa aplicação.

Vamos então criar este arquivo na mesma pasta dos outros artefatos que já criamos para o menu, com o nome `sidemenu-routing.module.ts` e vamos implementar nele o conteúdo a seguir.

```
import { SideMenuPage } from './sidemenu.page';
import { RouterModule, Routes } from '@angular/router';
import { NgModule } from '@angular/core';
import { OrdensDeServicoListagemPage } from
'../ordensdeservico/listagem/ordensdeservico-listagem.page';

const routes: Routes = [
  {
```

```
    path: 'menu',
    component: SideMenuPage,
    children: [
      {
        path: 'atendimentos',
        outlet: 'menucontent',
        component: OrdensDeServicoListagemPage,
      }
    ]
  },
  {
    path: '',
    redirectTo: '/menu/(menucontent:atendimentos)',
  }
];

@NgModule({
  imports: [
    RouterModule.forChild(routes)
  ],
  exports: [RouterModule]
})
export class SideMenuRoutingModule {}
```

Você notou a definição das rotas? Viu o `path` inicial como `menu`? Depois, temos o elemento `children`, que define um array, que por enquanto tem apenas um objeto. Este objeto define um novo `path`, que estará na continuidade do anterior, mas tem uma nova propriedade, o `outlet`, que define em qual `<ion-router-outlet>` a rota deverá ser renderizada. Como última propriedade, temos `component`, que define qual página será renderizada para a rota em questão.

O código termina com um `path` vazio, que é o inicial e que realiza o redirecionamento para a página principal da aplicação, que será exibida ao lado da página de menu. Em nosso caso, é a de listagem de ordens de serviços, que implementamos no final do capítulo anterior e que aqui chamamos de `atendimentos`.

A classe para o módulo do menu

Vamos criar nosso módulo na mesma pasta dos arquivos anteriores, agora, chamando `sidemenu.module.ts`, com o código que temos na sequência. Veja que ele é bem simples e dispensa comentários.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { IonicModule } from '@ionic/angular';

import { SideMenuPage } from './sidemenu.page';

import { SideMenuRoutingModule } from './sidemenu-routing.module';
import { OrdensDeServicoListagemPageModule } from
'../ordensdeservico/listagem/ordensdeservico-listagem.module';

@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    IonicModule,
    SideMenuRoutingModule,
    OrdensDeServicoListagemPageModule
  ],
  declarations: [SideMenuPage]
})
export class SideMenuPageModule {}
```

Apesar de termos implementado um novo sistema de rota, ele só será visível quando o componente de nosso menu renderizar a página. Então, precisamos chegar a ele quando a aplicação executar. E onde está o primeiro sistema de rotas? No `app-routing.module.ts`.

Podemos, nesse arquivo, comentar ou remover as rotas que já temos definidas e adicionar a que será responsável pelo menu. Veja-a na sequência.

```
{ path: '', loadChildren:
'./pages/menu/sidemenu.module#SideMenuPageModule' },
```

Para que nosso menu seja exibido, como já temos nossa página de ordens de serviço, que será exibida automaticamente, precisamos forçar a exibição do menu. Começamos isso com a injeção de `private menu: MenuController` em nosso construtor. Depois, em nosso `ionViewWillEnter()`, antes do final do método, vamos inserir `this.menu.open();`. Tudo isso em nosso `ordensdeservico-listagem.page.ts`.

Vamos então ver nossa aplicação em execução? Para ajudar, na sequência estão as instruções para o Android e iOS. Caso tenha problemas com o iOS, lembre-se da dica do capítulo anterior. Veja-a em execução na figura após as instruções

```
ionic cordova run android -l
ionic cordova run ios -l
```

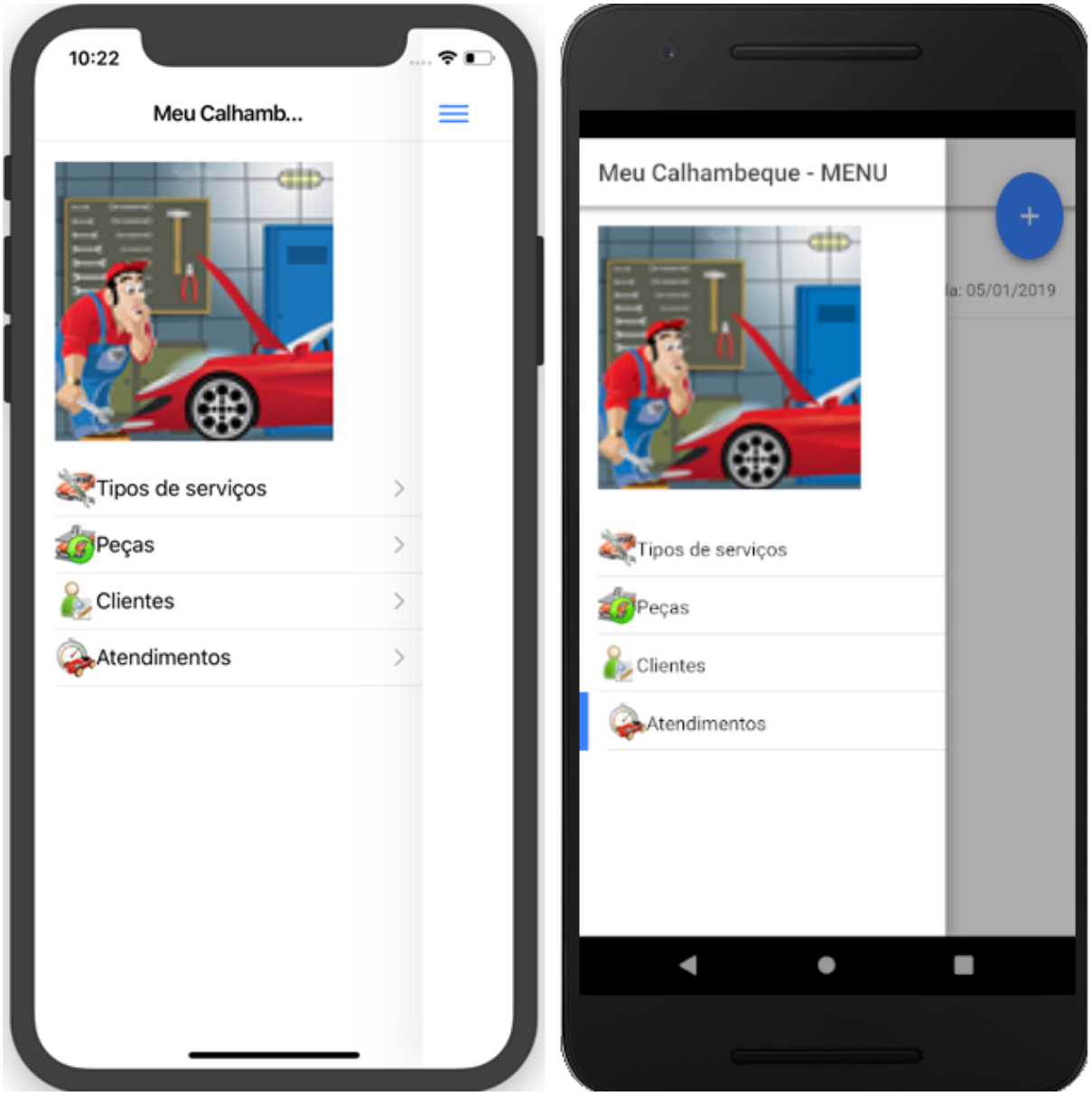


Figura 6.1: Menu em execução

6.2 A tabela de clientes na base de dados

No capítulo anterior, criamos um serviço específico para o acesso à base de dados, que foi implementado no arquivo `database.service.ts`. Neste serviço, temos o método `createTables()`,

onde criamos a tabela `ordensdeservico`. Precisamos, neste método, também criar a tabela `clientes` e o faremos com a inserção do código a seguir, logo antes da instrução que cria a tabela `ordensdeservico`, pois esta tabela terá um relacionamento com `clientes`, ao registrarmos um atendimento. Então, precisamos dela já criada.

```
await db.sqlBatch([
    ['CREATE TABLE IF NOT EXISTS clientes (clienteid TEXT primary key NOT
NULL, ' +
        'nome TEXT NOT NULL, email TEXT NOT NULL, telefone TEXT NOT NULL,
renda REAL NOT NULL, ' +
        'nascimento DATE NOT NULL)'],
    ['CREATE TABLE IF NOT EXISTS ordensdeservico (ordemdeservicoid TEXT
primary key NOT NULL, ' +
        'clienteid TEXT NOT NULL, veiculo TEXT NOT NULL, dataehoraentrada
DATETIME NOT NULL, ' +
        'dataehoratermino DATETIME, dataehoraentrega DATETIME)']
    ])
```

Com a tabela de clientes criada, precisamos populá-la, e faremos isso no método `populateDatabase()`. Antes de adaptarmos o método, eu extraí a instrução a seguir, que está dentro do `try{}` inicial do `INSERT` em `ordensdeservico` para logo no início do método, antes do `try{}`. Esta decisão foi tomada para que o registro inserido em `ordensdeservico` possua o mesmo valor para a coluna `clienteid` que foi inserida na tabela `clientes` como chave primária. Isso nos garantirá o relacionamento.

```
const clienteID = Guid.create().toString();
```

Agora sim, temos também, no código a seguir, nosso `INSERT` atualizado. Veja que, nos exemplos que dei, me preocupei em inserir apenas um registro para cada tabela.

```
await db.sqlBatch([
    ['insert into clientes ' +
        '(clienteid, nome, email, telefone, renda, nascimento) values ' +
        '(?, "Asbrusio", "asbrusio@cc.com", "999", 123.45, CURRENT_DATE)',
        [clienteID]
```

```
    ],
    ['insert into ordensdeservico ' +
        '(ordemdeservicoid, clienteid, veiculo, dataehoraentrada) values
(?, ?, ?, CURRENT_TIMESTAMP)',
        [ordemDeServicoID, clienteID, 'ABC-1235']]
])
```

Você deve ter verificado que, na criação da tabela `ordensdeservico` que fizemos no capítulo anterior, eu não me preocupei em efetivar o relacionamento entre as tabelas, mas é algo que deve ser pensado e, para resolver este problema, vamos reescrever o `CREATE TABLE` para `ordensdeservico`, tal qual está na sequência. Observe a instrução `FOREIGN KEY`.

```
['CREATE TABLE IF NOT EXISTS ordensdeservico (ordemdeservicoid TEXT
primary key NOT NULL, ' +
    'clienteid TEXT NOT NULL, veiculo TEXT NOT NULL, dataehoraentrada
DATETIME NOT NULL, ' +
    'dataehoratermino DATETIME, dataehoraentrega DATETIME, ' +
    'FOREIGN KEY (clienteid) REFERENCES clientes (clienteid) ON DELETE
CASCADE ON UPDATE NO ACTION)']
```

6.3 Service para clientes

Com a tabela de `clientes` criada na seção anterior, podemos antecipar a visão da necessidade de manipular estes dados, que serão recuperados, inseridos, alterados ou removidos da tabela. Isso nos remete à necessidade de termos nossa classe de modelo de negócio para clientes, e a implementaremos em nossa pasta `/src/app/models`, com o arquivo chamando `cliente.model.ts`, de acordo com o código:

```
export interface Cliente {
    clienteid: string;
    nome: string;
    email: string;
    telefone: string;
```

```
    renda: number;
    nascimento: Date;
}
```

Precisamos agora implementar a classe de serviços para `clientes` e um método inicial que nos retorne todos os clientes registrados na tabela, para que, em nosso registro de atendimento, possamos selecionar o cliente do veículo que entá entrando na oficina.

Na pasta `services` crie um arquivo chamado `clientes.service.ts` e insira nele o código apresentado na sequência. Observe que o código é semelhante ao que implementamos no capítulo anterior para `ordensdeservico`. Temos apenas uma diferença, que é a cláusula `ORDER BY`, no `SELECT`.

```
import { Injectable } from '@angular/core';
import { DatabaseService } from './database.service';
import { Cliente } from '../models/cliente.model';

@Injectable({
    providedIn: 'root'
})
export class ClientesService {
  constructor(
      private databaseService: DatabaseService
  ) { }

  public async getAll() {
    try {
      const db = await this.databaseService.getDatabase();
      const sql = 'SELECT * FROM clientes ORDER BY nome';
      try {
        const data = await db.executeSql(sql, []);
        if (data.rows.length > 0) {
          // tslint:disable-next-line:prefer-const
          let clientes: Cliente[] = [];
          for (let i = 0; i < data.rows.length; i++) {
            const cliente = data.rows.item(i);
            clientes.push(cliente);
          }
```

```
          return clientes;
        } else {
          return [];
        }
      } catch (e) {
        return console.error(e);
      }
    } catch (e) {
      return console.error(e);
    }
  }
}
```

6.4 A inserção de ordens de serviço

Começaremos com a criação da pasta onde teremos os arquivos que a comporão e daremos a ela o nome `add-edit`. Crie-a dentro da pasta `pages/ordensdeservico` e dentro dela os arquivos que trataremos nas subseções que se seguem.

O template para a página de registro

Vamos começar pelo nosso template, o arquivo `HTML`. Crie-o na pasta `pages/ordensdeservico/add-edit`, com o nome `ordensdeservico-add-edit.page.html` e nele implemente o código a seguir.

```
<ion-header>
    <ion-toolbar>
        <ion-buttons slot="start">
            <ion-back-button
defaultHref="/menu/(menucontent:atendimentos)"></ion-back-button>
        </ion-buttons>
        <ion-title>Ordem de Serviço</ion-title>
    </ion-toolbar>
</ion-header>

<ion-content padding>
```

```
    <ion-item-divider color="light">
        <ion-grid no-padding>
            <ion-row no-padding>
                <ion-col col-3 align-self-center>
                    <ion-img style="width: 50px; height: 50px;"
src="assets/imgs/icon_atendimentos.png"></ion-img>
                </ion-col>
                <ion-col col-9>
                    <ion-label style="font-size: 30px">Dados do
atendimento</ion-label>
                </ion-col>
            </ion-row>
        </ion-grid>
    </ion-item-divider>

    <ng-container *ngIf="osForm">
        <form [formGroup]="osForm">
            <ion-item lines="none" [disabled]="!modoDeEdicao">
                <ion-label position="floating" class="label-input-
title">Cliente</ion-label>
                <ion-select placeholder="Selecione um cliente"
formControlName="clienteid" style="margin-left: 3%;">
                    <ion-select-option *ngFor="let cliente of clientes"
[value]="cliente.clienteid">{{cliente.nome}}</ion-select-option>
                </ion-select>
            </ion-item>

            <ion-item lines="none" [disabled]="!modoDeEdicao">
                <ion-label position="floating" class="label-input-
title">Veículo</ion-label>
                <ion-input formControlName="veiculo" type="text"
style="margin-left: 3%;"></ion-input>
            </ion-item>

            <ion-item lines="none" [disabled]="!modoDeEdicao">
                <ion-label position="floating" class="label-input-
title">Data entrada</ion-label>
                <ion-datetime style="margin-left: 3%;"
formControlName="dataentrada" displayFormat="DD/MM/YYYY" min="1967"
max="2020-10-31"></ion-datetime>
            </ion-item>
```

```
            <ion-item lines="none" [disabled]="!modoDeEdicao">
                <ion-label position="floating" class="label-input-
title">Hora entrada</ion-label>
                <ion-datetime style="margin-left: 3%;"
formControlName="horaentrada" displayFormat="HH:mm"></ion-datetime>
            </ion-item>

            <ion-button *ngIf="!modoDeEdicao" shape="round"
color="primary" expand="block" padding>
                Alterar dados
            </ion-button>
            <div *ngIf="modoDeEdicao" no-padding>
                <ion-grid>
                    <ion-row>
                        <ion-col col-6 >
                            <ion-button shape="round" color="success"
size="small" padding expand="block">
                                Gravar
                            </ion-button>
                        </ion-col>

                        <ion-col col-6>
                            <ion-button shape="round" color="warning"
size="small" padding expand="block">
                                Cancelar
                            </ion-button>
                        </ion-col>
                    </ion-row>
                </ion-grid>
            </div>
        </form>
    </ng-container>
</ion-content>
```

Uma primeira mudança no código anterior refere-se ao componente que representa o botão para retorno à página anterior. Veja-o em detalhe na sequência. Observe a URL alvo. É nossa listagem de ordens de serviço já registradas.

```
<ion-back-button defaultHref="/menu/(menucontent:atendimentos)"></ion-back-button>
```

Observe a variável `modoDeEdicao`, que trabalharemos no TypeScript, para definir e identificar o estado da aplicação em relação ao atendimento em tela.

A próxima novidade é o componente `<ion-select>`, que será o responsável por exibir os clientes já registrados, para que possamos selecionar o proprietário do veículo que está entrando na oficina. Esta pode não ser a melhor opção, pois o menos custoso para a aplicação seria termos uma janela de pesquisa, onde o usuário possa digitar uma parte do nome e então uma seleção com este valor seja realizada. Faremos isso mais à frente, mas a ideia aqui é a de apresentar este novo controle.

```
<ion-select placeholder="Selecione um cliente" formControlName="clienteid" style="margin-left: 3%;">
    <ion-select-option *ngFor="let cliente of clientes" [value]="cliente.clienteid">{{cliente.nome}}</ion-select-option>
</ion-select>
```

Temos um `*ngFor` percorrendo uma coleção de clientes, que logo conheceremos. A propriedade `value`, que está ligada à propriedade `clienteid` de cada objeto obtido da coleção, permitirá obtermos o valor do ID do cliente selecionado, assim como temos o nome do cliente, que será utilizado para exibição na janela que exibirá os clientes registrados.

A classe TypeScript para nosso template

Vamos então para a implementação de nossa classe, que será responsável pela renderização de nosso template HTML. Veja o código dela logo na sequência. Crie-a na pasta `pages/ordensdeservico/add-edit`, com o nome `ordensdeservico-add-edit.page.ts` para o arquivo. Como você poderá verificar, não temos nada de novo, são funcionalidades que já implementamos em capítulos anteriores.

```
import { Component, OnInit } from '@angular/core';
import { OrdemDeServico } from 'src/app/models/ordemdeservico.model';
import { FormGroup, FormBuilder, Validators } from '@angular/forms';
import { Guid } from 'guid-typescript';
import { Cliente } from 'src/app/models/cliente.model';
import { ClientesService } from 'src/app/services/clientes.service';

@Component({
    templateUrl: './ordensdeservico-add-edit.page.html'
})
export class OrdensDeServicoAddEditPage implements OnInit {
    private ordemDeServico: OrdemDeServico;
    public modoDeEdicao = false;
    public osForm: FormGroup;
    public clientes: void | Cliente[] = [];

    constructor(
        private formBuilder: FormBuilder,
        private clientesService: ClientesService
    ) {}

    async ngOnInit() {
    }

    async ionViewWillEnter() {
        const clientes = await this.clientesService.getAll();
        this.clientes = clientes;

        this.ordemDeServico = {ordemdeservicoid:
Guid.createEmpty().toString(), clienteid: Guid.createEmpty().toString(),
veiculo: '', dataehoraentrada: new Date() };
        this.modoDeEdicao = true;

        this.osForm = this.formBuilder.group({
            ordemdeservicoid: [this.ordemDeServico.ordemdeservicoid],
            clienteid: [this.ordemDeServico.clienteid,
Validators.required],
            veiculo: [this.ordemDeServico.veiculo, Validators.required],
            dataentrada:
[this.ordemDeServico.dataehoraentrada.toISOString(), Validators.required],
            horaentrada:
```

```
[this.ordemDeServico.dataehoraentrada.toISOString(), Validators.required],
            dataehoraentrada: ['']
        });
    }
}
```

A classe para o módulo de manutenção dos dados do atendimento

Pois bem. Vamos à etapa final, que se refere à classe da definição do módulo. Daremos a ela o nome `ordensdeservico-add-edit.module.ts`, e a criaremos na mesma pasta dos arquivos anteriores. Veja o código na sequência.

```
import { Routes, RouterModule } from '@angular/router';
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { IonicModule } from '@ionic/angular';
import { OrdensDeServicoAddEditPage } from './ordensdeservico-add-
edit.page';

const routes: Routes = [
  {
    path: '',
    component: OrdensDeServicoAddEditPage
  }
];

@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    IonicModule,
    RouterModule.forChild(routes),
    ReactiveFormsModule
  ],
  declarations: [OrdensDeServicoAddEditPage]
})
export class OrdensDeServicoAddEditPageModule {}
```

Precisamos testar nossa aplicação, mas precisamos realizar algumas implementações no que já temos e aponto isso a seguir.

1. No arquivo `ordensdeservico-listagem.page.html` : em nosso `<ion-fab>` , vamos inserir dentro da tag o código `href='/menu/(menucontent:os-add-edit/-1)'` retirando o `routerLink` , que aponta para uma rota que ainda não temos implementada.

2. Vamos inserir o código a seguir como um objeto da propriedade `children` , no arquivo `sidemenu-routing.module.ts` . Ao inserirmos o código, precisaremos importar o arquivo, mas isso o VSC faz para nós.

```
,
{
    path: 'os-add-edit/:id',
    outlet: 'menucontent',
    component: OrdensDeServicoAddEditPage
}
```

3. No `sidemenu.module.ts` , dentro do `imports` do `NgModule` , precisamos inserir a importação para o novo módulo, `OrdensDeServicoAddEditPageModule` , que também deve ser nos `imports` do início do arquivo.

Execute agora sua aplicação e compare com a figura a seguir. Ela traz a visão de inserção, acessível a partir da visão de listagens de ordens de serviços, pela interação com o botão de adicionar em nosso FAB. A figura já traz o `select` em execução, apresentando uma listagem de clientes, que, no caso da figura, tem só um.

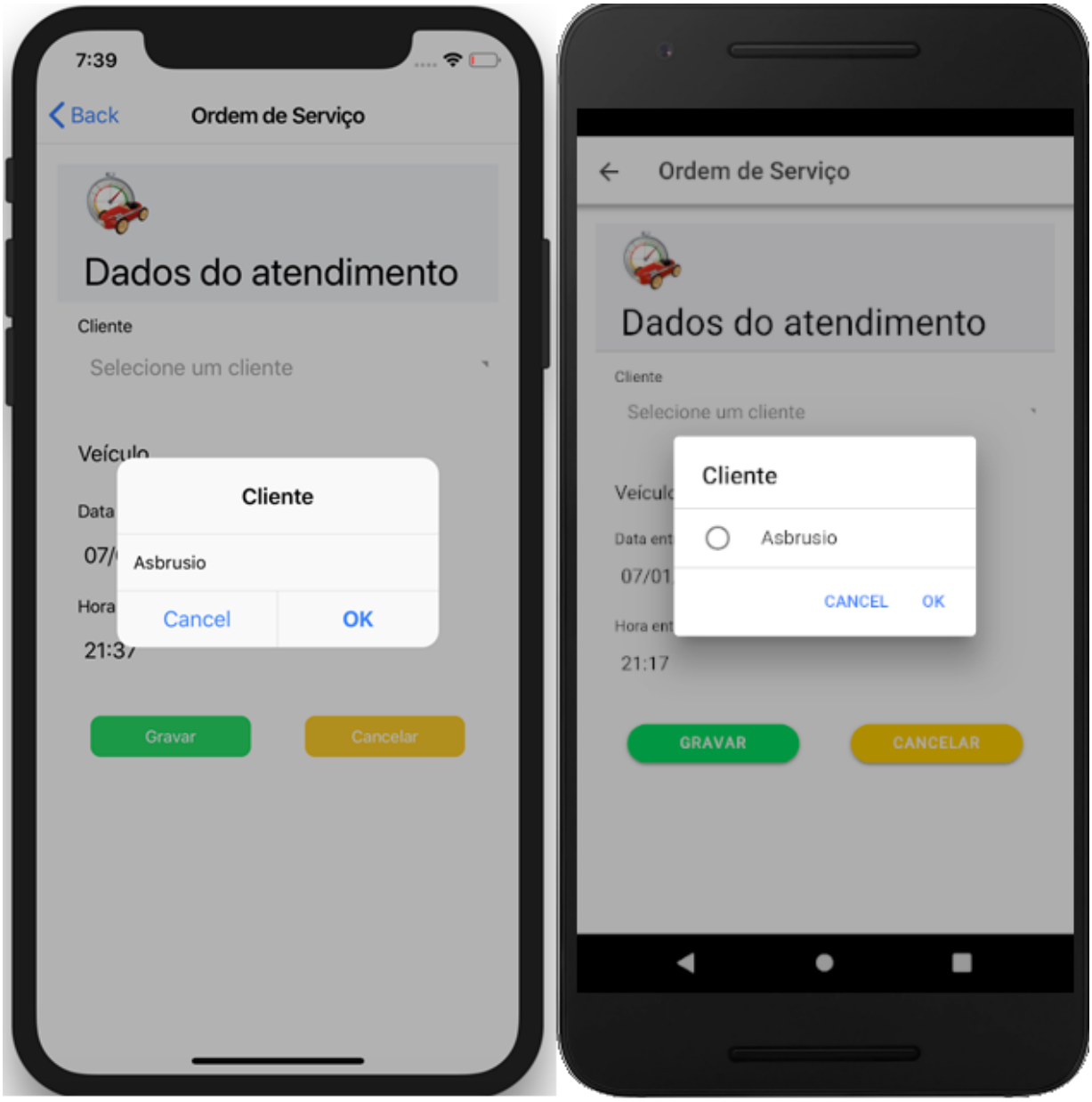


Figura 6.2: Página com a interface para inserção de ordens de serviço

6.5 Acesso a um atendimento existente

Nós temos a página de inserção de ordens de serviço pronta, mas nós ainda não registramos a inserção dela. Antes, quero que seja possível exibir a mesma visão para uma ordem de serviço já

existente, quando a selecionarmos na página de listagem. Precisamos realizar algumas alterações em nosso código. A primeira é em nossa listagem, no `<ion-item>` , dentro do `<ion-item-sliding>` . Precisamos incluir a URL de destino para quando o usuário interagir com um item da listagem. Veja a alteração no código a seguir.

```
<ion-item href='/menu/(menucontent:os-add-
edit/{{ordemDeServico.ordemdeservicoid}})'>
```

Com o identificador da ordem de serviço sendo enviado para nossa rota, nosso código para o template deverá estar preparado para recebê-lo e então recuperar a ordem de serviço da base de dados. Para isso, precisamos implementar um novo método em nosso `ordemdeservico.service.ts` , tal qual podemos ver na listagem a seguir.

```
public async getById(id: string): Promise<any> {
    try {
      const db = await this.databaseService.getDatabase();
      const sql = 'select * from ordensdeservico where ordemdeservicoid =
?';
      try {
        const data = await db.executeSql(sql, [ id ]);
        if (data.rows.length > 0) {
            const ordemdeservico: OrdemDeServico = data.rows.item(0);
            ordemdeservico.dataehoraentrada = new
Date(ordemdeservico.dataehoraentrada);
            return ordemdeservico;
        } else {
          return null;
        }
      } catch (e) {
        return console.error(e);
      }
    } catch (e) {
      return console.error(e);
    }
}
```

O código anterior, em relação ao `getAll()`, traz a cláusula `where` parametrizada, e este parâmetro é enviado ao `db.executeSql()`. Veja também que, após a verificação de recuperação do registro desejado, obtemos o primeiro item da coleção (índice 0). Como nossa propriedade `dataehoraentrada` é do tipo `Date`, precisamos converter o dado para este tipo antes de retornar o objeto recuperado.

Precisamos agora implementar a invocação dele em nosso código para o template. Para isso, vamos adaptar o `ionViewWillEnter()`, que já implementamos para a inserção, para o código que vemos na sequência. Estaremos aproveitando a implementação para torná-la funcional tanto para inserção como visualização dos dados da ordem de serviço selecionada na listagem.

```
async ionViewWillEnter() {
    const id = this.route.snapshot.paramMap.get('id');

    const clientes = await this.clientesService.getAll();
    this.clientes = clientes;

    const isIdEmptyGUID = Guid.parse(id).isEmpty();
    const isIdValidGUID = Guid.isGuid(id);

    if (id && !isIdEmptyGUID && isIdValidGUID) {
        this.ordemDeServico = await
this.ordensDeServicoService.getById(id);
    } else {
        this.ordemDeServico = {ordemdeservicoid:
Guid.createEmpty().toString(),
            clienteid: Guid.createEmpty().toString(), veiculo: '',
dataehoraentrada: new Date() };
        this.modoDeEdicao = true;
    }

    this.osForm = this.formBuilder.group({
        ordemdeservicoid: [this.ordemDeServico.ordemdeservicoid],
        clienteid: [this.ordemDeServico.clienteid, Validators.required],
        veiculo: [this.ordemDeServico.veiculo, Validators.required],
```

```
        dataentrada: [this.ordemDeServico.dataehoraentrada.toISOString(),
Validators.required],
        horaentrada: [this.ordemDeServico.dataehoraentrada.toISOString(),
Validators.required],
        dataehoraentrada: ['']
    });

    this.ordemDeServico = this.osForm.value;
}
```

Notou que recuperamos o `id`, fazendo uso de rotas? Precisaremos injetar no construtor a propriedade `private route: ActivatedRoute` para isso. Depois, recuperamos os dados dos clientes registrados, pois precisaremos ter a coleção disponibilizada, tanto para selecionar um novo cliente, se for o caso, como para exibir o nome do cliente registrado para o atendimento.

Em seguida, verificamos se o `id` recebido refere-se a um já existente, para então recuperarmos seus respectivos dados da base com nosso novo método `getById()`. Caso o `id` não seja um já existente, é porque o usuário decidiu registrar um novo. O registro do formulário é o mesmo que vimos para a inserção. Para a invocação deste método, precisamos injetar no construtor a propriedade `private ordensDeServicoService: OrdensDeServicoService`.

Observe ainda que, como nosso objeto que será exibido na visão é diferente daquele que recuperamos e com o qual populamos nosso `osForm`, precisamos trazer para o `this.ordemDeServico` o objeto que será utilizado por nosso template HTML.

Precisamos também adaptar nosso código de template para o elemento `<ion-select-option>`, onde deveremos adicionar `[selected]="cliente.clienteid===ordemDeServico.clienteid"`, para que possa ser exibido o nome do cliente já registrado no atendimento. Agora podemos testar nossa aplicação, tanto para inserção como para seleção de um atendimento já registrado.

6.6 Gravação dos dados do atendimento

Temos uma única página para manipulação das ordens de serviço, que atende tanto à inserção, quanto à atualização dos dados. Você deve ter notado que, quando executamos a inserção, o botão para gravação está habilitado. Entretanto, quando selecionamos um atendimento na listagem, os dados aparecem desabilitados e temos um botão com o texto `Alterar dados`, que habilitará os controles para atualização, caso esta seja a necessidade do usuário.

Com esta contextualização, podemos identificar dois pontos que precisamos implementar: a liberação para alteração dos dados e a gravação deles, que são atividades respectivas aos botões comentados anteriormente e, aproveitaremos para implementar também o que cancela a edição, retornando os dados à sua última versão gravada.

Entretanto, antes de partirmos para estas implementações, vamos abstrair o comportamento da gravação. Precisaremos atualizar o registro na base de dados, ou inserir o novo. Isso nos leva a implementar um serviço em nosso `ordensdeservico.service.ts`, que vamos chamar de `update()`, seguindo a mesma nomenclatura que adotamos para peças. Veja o código a seguir.

```
async update(ordemdeservico: OrdemDeServico): Promise<OrdemDeServico> {
    let sql: any;
    let params: any;

    if (Guid.parse(ordemdeservico.ordemdeservicoid).isEmpty()) {
      ordemdeservico.ordemdeservicoid = Guid.create().toString();
      sql = 'INSERT INTO ordensdeservico(ordemdeservicoid, clienteid,
veiculo, dataehoraentrada) ' +
        'values(?, ?, ?, ?)';
      params = [ordemdeservico.ordemdeservicoid, ordemdeservico.clienteid,
        ordemdeservico.veiculo, ordemdeservico.dataehoraentrada ];
    } else {
      sql = 'UPDATE ordensdeservico SET clienteid = ?, veiculo = ?, ' +
        'dataehoraentrega = ? WHERE ordemdeservicoid = ?';
```

```
      params = [ordemdeservico.clienteid,
        ordemdeservico.veiculo, ordemdeservico.dataehoraentrada,
ordemdeservico.ordemdeservicoid ];
    }

    try {
      const db = await this.databaseService.getDatabase();
      try {
        await db.executeSql(sql, params);
        return ordemdeservico;
      } catch (e) {
        console.error(e);
      }
    } catch (e) {
      console.error(e);
    }
}
```

Começamos declarando duas variáveis que serão responsáveis por conter a instrução SQL que será executada e os parâmetros que deverão ser utilizados nessa instrução. Em seguida, com base no valor de `ordemdeservico.ordemdeservicoid`, caso seja um GUID, verificamos que instrução SQL devemos preparar para a execução. Veja que, no caso de inserção, criamos um GUID para a propriedade. Após esta definição, executamos o método por meio da chamada à `db.executeSql()`.

Com o serviço devidamente implementado, vamos retomar a linha de pensamento que nos levou a ele, que se refere à implementação dos métodos de gravação, cancelamento de edição e liberação da edição. Veja os métodos que precisaremos implementar no arquivo `ordensdeservico-add-edit.page.ts` na sequência.

```
// Método a ser invocado quando o botão de alterar dados for selecionado
iniciarEdicao() {
    this.modoDeEdicao = true;
}

// Método a ser invocado quando o botão de cancelar alteração for
selecionado
```

```
cancelarEdicao() {
    this.osForm.setValue(this.ordemDeServico);
    this.modoDeEdicao = false;
}

// Método a ser invocado quando o botão de gravar for selecionado
async submit() {
// Validação dos dados informados no formulário. Já trabalhamos com isso.
    if (this.osForm.invalid || this.osForm.pending) {
        await this.alertService.presentAlert('Falha', 'Gravação não foi
executada',
            'Verifique os dados informados para o atendimento', ['Ok']);
        return;
    }

// Aqui extraímos a data e hora da informadas no formulário e convertemos
para um Date
    const dataString = new
Date(this.osForm.controls.dataentrada.value).toDateString();
    const horaString = new
Date(this.osForm.controls.horaentrada.value).toTimeString();
    const dataEHora = new Date(dataString + ' ' + horaString);

// Invocamos o serviço, enviando um objeto com os dados recebidos da visão
    this.ordemDeServico = await this.ordensDeServicoService.update(
        {
            ordemdeservicoid: this.osForm.controls.ordemdeservicoid.value,
            clienteid: this.osForm.controls.clienteid.value,
            veiculo: this.osForm.controls.veiculo.value,
            dataehoraentrada: dataEHora,
        }
    );

// Informamos o usuário do sucesso da operação e o redirecionamos para a
listagem
    this.toastService.presentToast('Gravação bem sucedida', 3000, 'top');
    this.router.navigateByUrl('/menu/(menucontent:first)');
}
```

Notou o uso da variável `modoDeEdicao` como responsável para definição do estado atual da aplicação em relação ao atendimento

sendo exibido ao usuário?

Para a correta compilação do código anterior, precisamos injetar alguns objetos no construtor, que são: `private toastService: ToastService`, `private alertService: AlertService` e `private router: Router`.

Veja que estamos injetando um `AlertService`, que ainda não temos, mas trago na sequência o código que deverá ser implementado em um arquivo chamado `alert.service.ts`, na pasta `services`.

```
import { Injectable } from '@angular/core';
import { AlertController } from '@ionic/angular';

@Injectable({
    providedIn: 'root'
  })
export class AlertService {
    constructor(private alertCtrl: AlertController) {}

    async presentAlert(header: string, subHeader: string, message: string,
buttons: string[]) {
        const alert = await this.alertCtrl.create({
          header,
          subHeader,
          message,
          buttons
        });
        await alert.present();
    }
}
```

No processo de redirecionamento, poderíamos pensar que, se estivéssemos em uma inserção, seria interessante ficar na mesma página, para que uma nova pudesse ocorrer, mas isso fica com você, já temos a lógica e ferramental para isso.

Ainda não está pronto. Temos os métodos implementados, mas precisamos atribuí-los em nosso template, nos controles, para que o efeito desejado pelo usuário na interação com os botões possa

ocorrer. No botão de alterar dados, insira `(click)="iniciarEdicao()"` e, no de cancelar, insira `(click)="cancelarEdicao()"` .

Para a gravação, precisamos realizar duas alterações. A primeira é na definição do `<form>` , onde devemos inserir `(submit)="submit()"` , que liga nosso método ao processo de submissão do formulário. A segunda, agora sim, no botão, é apenas inserir `type="submit"` no código da tag do botão de gravação.

Agora podemos testar nossa aplicação para inserirmos um novo atendimento e selecionarmos um que seja exibido na listagem de atendimentos registrados.

6.7 Removendo um atendimento já registrado

Falta-nos a remoção de um atendimento, que é o que faremos nesta seção. Sabemos que é preciso um método em nosso arquivo `ordensdeservico.service.ts` e podemos vê-lo na listagem a seguir.

```
async removeById(id: string): Promise<boolean | void> {
  try {
    await this.databaseService.getDatabase()
      .then( async (db) => await db.executeSql('DELETE FROM
ordensdeservico WHERE ordemdeservicoid = ?', [ id ])
      .then( () => true));
  } catch (e) {
    throw new Error(e);
  }
}
```

Você notou que estamos optando pelo uso de `then()` também na invocação de `getDatabase()` ? Isso é uma boa estratégia quando temos duas invocações para chamadas assíncronas no mesmo método, onde uma depende da execução de outra. Veja também que no `generics` para o tipo de retorno estamos informando dois

tipos de dados, o `boolean`, que é para o sucesso, e `void`, para o caso de exceção.

Com o serviço pronto, precisamos implementar a invocação a ele em nosso arquivo `ordensdeservico-listagem.page.ts`, pois a exclusão será possível por meio da listagem, com a interação em menu que aparecerá quando o usuário deslizar um item. Veja esse método na sequência e note que precisamos injetar o `AlertService`.

```
async removerAtendimento(ordemdeservico: OrdemDeServico) {
    await
this.ordensdeservicoService.removeById(ordemdeservico.ordemdeservicoid)
        .then( async () => {
            this.ordensDeServico = await
this.ordensdeservicoService.getAll();
            this.toastService.presentToast('Ordem de Serviço removida',
3000, 'top');
            await this.slidingList.closeSlidingItems();
        })
        .catch( async (e) =>  await
this.alertService.presentAlert('Falha', 'Remoção não foi executada', e,
['Ok']));
}
```

Agora, em nosso template, precisamos inserir o código para exibir o botão de remoção e, neste código, temos a requisição ao método que implementamos anteriormente.

```
<ion-item-options side="end">
    <ion-item-option color="danger"
(click)="removerAtendimento(ordemDeServico)">
      <ion-button color="danger" no-padding>
          <ion-icon name="trash"></ion-icon>
          Remover
      </ion-button>
    </ion-item-option>
</ion-item-options>
```

Podemos testar nossa aplicação e remover um atendimento de nossa listagem. Não apresentarei figuras, pois é semelhante ao que

já vimos.

6.8 Nossa implementação com o Capacitor

Como dito no capítulo 2, o Capacitor tem se apresentado como vanguarda, como substituto evolutivo do Cordova, ganhando espaço. Podemos utilizar plugins do Cordova também para o Capacitor e é o que faremos agora. Vamos a algumas instruções iniciais para este processo e então testaremos nossa aplicação agora nele. Alguns comentários são apresentados entre as instruções, que devem ser executadas no terminal.

```
// Preparação para o Capacitor
ionic integrations enable capacitor
npm run build

// Instalação de componenes capacitor e remoção do cordova
npm install --save @capacitor/cli @capacitor/core
npm uninstall --save cordova-plugin-splashscreen

// Inicialização do capacitor já definindo o nome e package para a
aplicação
npx cap init IonicCC br.com.casadocodigoionic

// Adição das plataformas de execução da aplicação no capacitor
npx cap add ios
npx cap add android

// Abre o Android Studio e XCode para a execução da aplicação. Nos
capítulos anteriores já vimos como utilizar com sincronização.
npx cap open ios
npx cap open android
```

Conclusão

Terminamos este interessante capítulo. Não vimos muitos recursos do Ionic nele, mas nos dedicamos a técnicas e serviços, e

concluímos um CRUD com uma base de dados local à aplicação, o SQLite. Em nosso próximo capítulo, faremos uso do Firestore, que, de maneira resumida, é uma base de dados com base em documentos, disponibilizada remotamente, na nuvem. Também implementaremos uma página específica para pesquisa, que substituirá nosso `<ion-select>`. Será um capítulo muito legal. Já nos vemos!

Capítulo 7
CRUD com Firestore para acesso a dados remoto e uma página de pesquisa

No capítulo anterior adaptamos nossa aplicação para uma "cara" mais próxima de um aplicativo móvel, com o Sidemenu e manipulação de dados permanentes, em uma base de dados SQLite. Vimos o componente `<ion-select>`, que nos apresentou uma coleção de dados, onde elegemos um para compor os dados de um atendimento a um veículo.

Entretanto, as aplicações móveis, em relação aos dados manuseados por ela, fazem uso cada vez mais frequente e restrito dos dados remotos, inclusive, em alguns casos, sem a necessidade de sincronização com uma base local. E é isso que trabalharemos neste capítulo, por meio do Firestore, que apresentarei mais adiante com definições e características.

Com exceção do acesso ao Firestore e das instruções para manipulação dos dados, nossas implementações neste capítulo trabalharão com técnicas semelhantes às que estamos vendo desde o capítulo 5. Mas veremos também uma solução mais indicada para seleção de um dado para utilização em um formulário, em substituição ao `<ion-select>`. Criaremos uma página específica para pesquisa de dados.

7.1 Preparação do projeto para o Firestore

De acordo com Jorge Vergara (https://blog.ionicframework.com/building-ionic-apps-with-firestore/), o Firestore é um banco de dados baseado em documentos NoSQL, totalmente gerenciado para dispositivos móveis e web. Ele é

projetado para armazenar e sincronizar dados de aplicativos com facilidade. Ou seja, é um produto que lida diretamente com aplicações para dispositivos móveis, sem necessidades de nos preocuparmos com camadas além da nossa aplicação.

O Firestore teve sua origem no Firebase, ambos produtos do Google e, caso queira, você pode obter um texto específico para a tecnologia em https://firebase.google.com/docs/firestore/.

Falamos anteriormente sobre documentos NoSQL e que o Firestore é uma base de dados que tem como estrutura estes documentos, esse paradigma tende a ser mais específico e mais simples.

Mas o que é NoSQL? O que são documentos? O NoSQL trata as tabelas como coleções e seus registros como documentos, que são objetos baseados em entradas específicas a um domínio de dados. Uma grande diferença que surpreende os iniciantes é que uma base relacional, como a que vimos nos capítulos anteriores, precisa de estruturas predefinidas para ser populada. Se um novo dado surgir, precisamos alterar a estrutura das tabelas, para receber este dado. Com NoSQL, essa preocupação deixa de existir, é dinâmica, cada objeto da coleção tem sua estrutura atrelada aos dados que o compõem. Os tipos de dados possíveis para cada objeto são muitos, podendo inclusive uma coleção fazer parte de outra.

Instalação dos plugins necessários

Após ter decidido por criar um novo projeto ou trabalhar no do capítulo anterior, acesse o terminal, na pasta do projeto, e execute a instrução a seguir.

```
npm install @angular/fire@5.2.1 firebase@6.1.1
```

Com os plugins devidamente instalados, precisamos acessar o portal do `Firebase Console`, por meio do link https://console.firebase.google.com. Por ser um produto Google, é importante que tenhamos uma conta do Gmail para realizarmos o acesso à plataforma.

Com o acesso devidamente realizado, o serviço oferece a opção de `Adicionar Projeto`. Forneça um nome para seu projeto, a ferramenta verificará a disponibilidade para ele. Configure as localizações solicitadas de acordo com sua necessidade. Eu optei por `Brasil` para o `Analytics` e `us-central` para o Firestore. Ao final da interface confirme o aceite para os termos de acesso e a criação do projeto. Você será informado quando a criação for concluída.

A página que será exibida após a confirmação da mensagem de conclusão da criação do projeto é conhecida por `Get Started` do `Project Overview`, que tem um link de acesso na barra lateral do `Firebase Console`. Esta página oferece alguns botões relacionados às plataformas, `iOS`, `Android`, `Web </>` e `Unit`. Nós confirmaremos a criação para uma aplicação web, lembrando que nossas aplicações Ionic são executadas em um ambiente web, por padrão.

Uma janela popup aparecerá, com um código JavaScript, que deve ser copiado, pois usaremos estas informações em nosso código. Para efeito ilustrativo, este código é semelhante ao apresentado na sequência.

```
<script src="https://www.gstatic.com/firebasejs/5.7.3/firebase.js">
</script>
<script>
  // Initialize Firebase
  var config = {
    apiKey: "BIzaSyCNi6RyHWWd2vNtfGDZSolWVPEI41fr3vc",
    authDomain: "ionicccfb.firebaseapp.com",
    databaseURL: "https://ionicccfb.firebaseio.com",
    projectId: "ionicccfb",
    storageBucket: "ionicccfb.appspot.com",
    messagingSenderId: "80336637376"
  };
  firebase.initializeApp(config);
</script>
```

Como nossa aplicação não é uma aplicação web comum, não podemos copiar o código no arquivo HTML, como sugere a janela comentada anteriormente. Precisamos, em nosso projeto Ionic, na

pasta `/src/app`, criar o arquivo `credentials.ts` e usar parte do código anterior atrelado à sintaxe do JavaScript. Veja este código na sequência. Com a exportação, poderemos importar este objeto em qualquer ponto de nossa aplicação.

```
export const firebaseConfig = {
    apiKey: "BIzaSyCNi6RyHWWd2vNtfGDZSolWVPEI41fr3vc",
    authDomain: "ionicccfb.firebaseapp.com",
    databaseURL: "https://ionicccfb.firebaseio.com",
    projectId: "ionicccfb",
    storageBucket: "ionicccfb.appspot.com",
    messagingSenderId: "80336637376"
};
```

Com nossa configuração implementada no projeto, precisamos iniciar o Firebase em nosso já conhecido arquivo `app.module.ts`, começando com a importação dos pacotes `AngularFire2` que instalamos anteriormente. Veja as instruções que devemos adicionar no arquivo a seguir.

```
import { AngularFireModule } from '@angular/fire';
import { AngularFirestoreModule } from '@angular/fire/firestore';
import { firebaseConfig } from './credentials';
```

Precisamos importar para nosso módulo os módulos referenciados anteriormente e fazemos isso inserindo as instruções a seguir no `imports` dentro de nosso `@NgModule()`. Veja que, ao chamarmos o `AngularFireModule.initializeApp(firebaseConfig)`, estamos enviando nossas credenciais para a inicialização da aplicação, o que fará com que nossa aplicação conecte-se com o `Firebase`.

```
AngularFireModule.initializeApp(firebaseConfig),
AngularFirestoreModule
```

7.2 Populando a base de dados do Firestore

No capítulo anterior, nós trabalhamos um CRUD para `Ordens de Serviço`, onde fazíamos uso de uma associação/relacionamento com `Clientes`, mas tínhamos dados previamente inseridos na tabela. Agora vamos implementar o CRUD para `Clientes` e, diferente das implementações anteriores, para podermos variar um pouco, vamos começar pela inserção de clientes. Já sabemos que temos uma série de burocracias para termos este serviço disponibilizado em nossa aplicação e faremos isso nas subseções a seguir.

O template para a página de registro

Em `/src/app/pages`, vamos criar a pasta `clientes` e nela, outra, chamada `add-edit`. Com esta estrutura preparada, vamos ao nosso arquivo. Crie-o com o nome `clientes-add-edit.page.html` e nele implemente o código a seguir.

```
<ion-header>
    <ion-toolbar>
        <ion-buttons slot="start">
            <ion-back-button defaultHref="/menu/(menucontent:clientes)">
</ion-back-button>
        </ion-buttons>
        <ion-title>Clientes</ion-title>
    </ion-toolbar>
</ion-header>

<ion-content padding>
    <ion-item-divider color="light">
        <ion-grid no-padding>
            <ion-row no-padding>
                <ion-col col-3 align-self-center>
                    <ion-img style="width: 50px; height: 50px;"
src="assets/imgs/icon_clientes.png"></ion-img>
                </ion-col>
                <ion-col col-9>
                    <ion-label style="font-size: 30px">Dados do
cliente</ion-label>
                </ion-col>
            </ion-row>
```

```
        </ion-grid>
    </ion-item-divider>

    <ng-container *ngIf="clientesForm">
        <form [formGroup]="clientesForm">
            <ion-item lines="none" [disabled]="!modoDeEdicao">
                <ion-label position="floating" class="label-input-
title">Nome</ion-label>
                <ion-input formControlName="nome" type="text"
style="margin-left: 3%;"></ion-input>
            </ion-item>

            <ion-item lines="none" [disabled]="!modoDeEdicao">
                <ion-label position="floating" class="label-input-
title">Email</ion-label>
                <ion-input formControlName="email" type="email"
style="margin-left: 3%;"></ion-input>
            </ion-item>

            <ion-item lines="none" [disabled]="!modoDeEdicao">
                <ion-label position="floating" class="label-input-
title">Telefone</ion-label>
                <ion-input formControlName="telefone" type="tel"
style="margin-left: 3%;"></ion-input>
            </ion-item>

            <ion-item lines="none" [disabled]="!modoDeEdicao">
                <ion-label position="floating" class="label-input-
title">Renda</ion-label>
                <ion-input formControlName="renda" type="number"
style="margin-left: 3%;"></ion-input>
            </ion-item>

            <ion-item lines="none" [disabled]="!modoDeEdicao">
                <ion-label position="floating" class="label-input-
title">Nascimento</ion-label>
                <ion-datetime style="margin-left: 3%;"
formControlName="nascimento" displayFormat="DD/MM/YYYY" min="1967"
max="2020-10-31"></ion-datetime>
            </ion-item>
```

```
            <ion-button *ngIf="!modoDeEdicao" shape="round"
color="primary" expand="block" padding>
                Alterar dados
            </ion-button>
            <div *ngIf="modoDeEdicao" no-padding>
                <ion-grid>
                    <ion-row>
                        <ion-col col-6 >
                            <ion-button shape="round" color="success"
size="small" padding expand="block">
                                Gravar
                            </ion-button>
                        </ion-col>

                        <ion-col col-6>
                            <ion-button shape="round" color="warning"
size="small" padding expand="block">
                                Cancelar
                            </ion-button>
                        </ion-col>
                    </ion-row>
                </ion-grid>
            </div>
        </form>
    </ng-container>
</ion-content>
```

A classe TypeScript para nosso template

Vamos então para a implementação de nossa classe, que, como já sabemos, é responsável pela renderização de nosso template HTML. Veja o código dela logo na sequência. Crie-a na pasta `pages/clientes/add-edit` , com o nome `clientes-add-edit.page.ts` para o arquivo. De início, o objeto `cliente` está inicializado como novo, pois nosso objetivo agora é a criação de um novo cliente. Precisaremos trazer nosso arquivo `cliente.model.ts` para nosso projeto, caso você tenha criado um projeto do início.

```
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormBuilder, Validators } from '@angular/forms';
```

```
import { Cliente } from 'src/app/models/cliente.model';
import { MenuController } from '@ionic/angular';

@Component({
    templateUrl: './clientes-add-edit.page.html'
})
export class ClientesAddEditPage implements OnInit {
    private cliente: Cliente;
    public modoDeEdicao = false;
    public clientesForm: FormGroup;

    constructor(
        private formBuilder: FormBuilder
    ) {}

    async ngOnInit() {
    }

    async ionViewWillEnter() {
        this.modoDeEdicao = true;

        this.cliente = {clienteid: '', nome: '', email: '', telefone: '',
renda: 0.00, nascimento: new Date()};

        this.clientesForm = this.formBuilder.group({
            clienteid: [this.cliente.clienteid],
            nome: [this.cliente.nome, Validators.required],
            email: [this.cliente.email, Validators.required],
            telefone: [this.cliente.telefone, Validators.required],
            renda: [this.cliente.renda, Validators.required],
            nascimento: [this.cliente.nascimento.toISOString(),
Validators.required]
        });
    }
}
```

A classe para o módulo de manutenção dos dados do cliente

Pois bem. Vamos à etapa final, que se refere à classe onde definimos o módulo. Seguindo nossa convenção, daremos a ela o

nome `clientes-add-edit.module.ts`, e a criaremos na mesma pasta dos arquivos anteriores. Veja o código na sequência.

```
import { Routes, RouterModule } from '@angular/router';
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { IonicModule } from '@ionic/angular';
import { ClientesAddEditPage } from './clientes-add-edit.page';

const routes: Routes = [
  {
    path: '',
    component: ClientesAddEditPage
  }
];

@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    IonicModule,
    RouterModule.forChild(routes),
    ReactiveFormsModule
  ],
  declarations: [ClientesAddEditPage]
})
export class ClientesAddEditPageModule {}
```

Já temos a interface com o usuário pronta. Agora precisamos testar nossa aplicação. Só que, para isso, tal qual fizemos para os atendimentos no capítulo anterior, precisamos realizar algumas implementações no que já temos:

1. Vamos inserir o código a seguir como um objeto da propriedade `children`, no arquivo `sidemenu-routing.module.ts`. Ao inserirmos o código, precisaremos importar o arquivo. Aqui também teremos que trazer componentes do capítulo anterior, o nosso `sidemenu`, caso você tenha criado um projeto do início. Você poderá

comentar os códigos para atendimentos ou já importar também tudo que for necessário.

```
,
{
    path: 'clientes-add-edit',
    outlet: 'menucontent',
    component: ClientesAddEditPage
}
```

2. No `sidemenu.module.ts`, dentro do `imports` do `NgModule`, precisamos inserir a importação para o novo módulo, `ClientesAddEditPageModule`, que também deve ser inserido nos `imports` do início do arquivo.

3. Em nosso `sidemenu.page.ts`, precisamos alterar o valor da `url` de clientes, para que ao selecionarmos a opção de clientes na página inicial o redirecionamento para o `clientes-add-edit` possa ocorrer para a página que estamos concluindo.

4. No `app.component.ts`, optei por retirar a invocação ao serviço de criação da base de dados, pois neste capítulo não o utilizaremos e isso pode gerar diversos erros no console do navegador, uma vez que os serviços relacionados à persistência, vistos no capítulo anterior, não executam no navegador.

5. Novamente no `sidemenu-routing.module.ts`, precisamos configurar agora a URL para redirecionamento na primeira execução, tal qual podemos ver na sequência:

```
{
    path: '',
    redirectTo: '/menu/(menucontent:clientes-add-edit)',
}
```

6. Temos que mudar a rota inicial em nosso `app-routing.module.ts` para o que temos na sequência.

```
{ path: '', loadChildren:
'./pages/menu/sidemenu.module#SideMenuPageModule' },
```

Execute a aplicação e compare com as figuras a seguir. Elas trazem a visão de inserção, mas não temos ainda nenhum comportamento em relação à interação com o usuário.

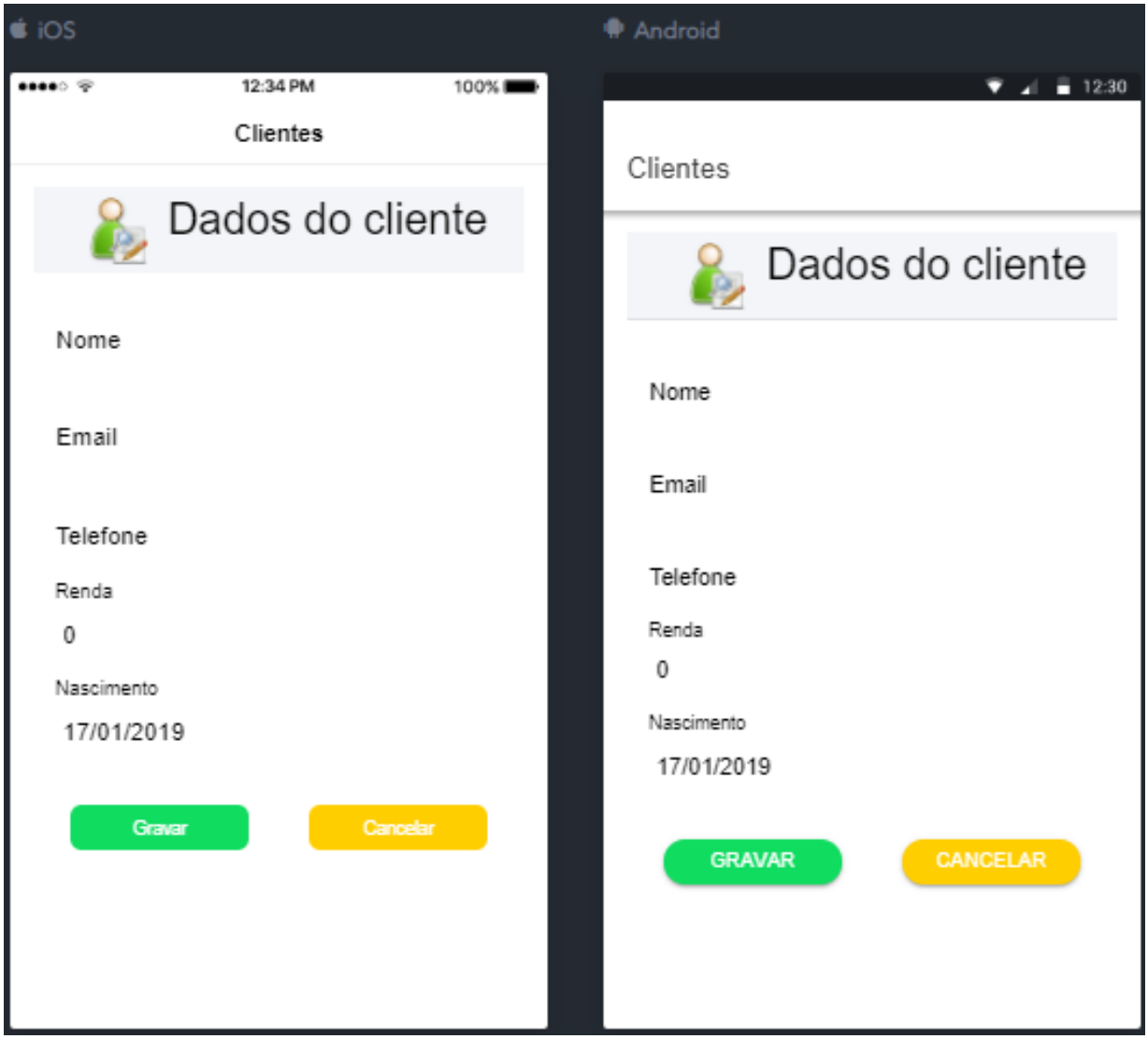


Figura 7.1: Página com a interface para inserção de clientes

Gravação dos dados do cliente

Seguiremos para clientes a mesma metodologia dos exemplos anteriores. Teremos uma única visão para visualização dos dados, alteração e inserção. Desta maneira, as implementações trabalhadas aqui já terão como foco esta característica.

Como estamos trabalhando a situação de criação de um novo cliente, precisamos nos preocupar com a criação do valor para a propriedade `clienteid`, que é identificadora de cada objeto. O `Firestore` nos traz um recurso interessante: ele mesmo nos fornece um identificador único e faremos uso dele.

Em nossa pasta `/src/app/services`, vamos criar um arquivo que manterá serviços globais que precisemos consumir. Chame este arquivo de `firestore.service.ts` e implemente o código a seguir nele.

```
import { Injectable } from '@angular/core';
import { AngularFirestore } from '@angular/fire/firestore';

@Injectable({
    providedIn: 'root'
})
export class FirestoreService {
    constructor(private firestore: AngularFirestore) {}

    createID(): string {
        return this.firestore.createId();
    }
}
```

Agora vamos para a implementação específica de um serviço para clientes. O primeiro código que trabalharemos é em nossa classe de serviço, do arquivo `clientes.service.ts`, que já temos implementado do capítulo anterior, onde recuperamos os clientes e populamos nosso `<ion-select>`. Mantenha o código que já tem, ou apenas comente o comportamento dele, deixando a assinatura, para evitarmos problemas de compilação. Vamos a ele, veja a listagem a seguir.

```
import { Injectable } from '@angular/core';
import { Cliente } from '../models/cliente.model';
import { AngularFirestore, AngularFirestoreCollection } from
'@angular/fire/firestore';

@Injectable({
    providedIn: 'root'
})
export class ClientesService {
  constructor(
    private firestore: AngularFirestore
  ) { }

  async update(cliente: Cliente): Promise<void> {
    try {
      cliente.nascimento = new Date(cliente.nascimento);
      await
this.firestore.doc(`clientes/${cliente.clienteid}`).set(cliente);
    } catch (e) {
      console.error(e);
    }
  }
}
```

Identificou de imediato a injeção de um objeto `AngularFirestore` ? É por meio deste objeto que nos comunicaremos com o serviço para atualizarmos e recuperarmos os dados necessários para nossa aplicação. Depois, na implementação de `update()` , temos a chegada de um `Cliente` e o retorno padrão para o serviço que consumiremos, que é um `Promise<void>` . No bloco `try` temos a invocação do método `set()` , com o `cliente` que recebemos, mas, antes dele, convertemos o dado que chegou como string para `Date` . Observe que este método, o `set()` , é uma chamada encadeada ao método `doc()` , que tem uma URI para nossa coleção e, por meio do `id` , enviado nessa URI, identifica a qual documento a operação se refere.

Vamos para o método `submit()` , de nosso arquivo que renderiza o template de nossa página. Verifique, no código a seguir, a

simplicidade de compreensão do código.

```
async submit() {
    if (this.clientesForm.invalid || this.clientesForm.pending) {
        await this.alertService.presentAlert('Falha', 'Gravação não foi
executada',
            'Verifique os dados informados para o atendimento', ['Ok']);
        return;
    }

    // Estamos utilizando pela primeira vez este controle
    const loading = await this.loadingCtrl.create();
    await loading.present();

    if (this.cliente.clienteid === '') {

this.clientesForm.controls.clienteid.setValue(this.firestoreService.create
ID());
    }

    // Invocamos o serviço, enviando um objeto com os dados recebidos da
visão
    await this.clientesService.update(this.clientesForm.value)
        .then(
            () => {
                loading.dismiss().then(() => {
                    this.toastService.presentToast('Gravação bem
sucedida', 3000, 'top');
                });
            },
                error => {
                console.error(error);
            }
        );
}
```

No código anterior, após a validação do formulário, temos o uso de um novo componente, um `LoadingController`, que devemos injetar no construtor, com a instrução `private loadingCtrl: LoadingController`. Daremos ao usuário o sentimento de que a aplicação está

processando uma requisição, evitando a impressão de congelamento de tela. Em seguida, obtemos um `ID`, criado pelo `Firestore` e o atribuímos ao objeto. Para usarmos o `update()`, precisamos de nosso `clienteService` disponibilizado, sendo assim, vamos ao código para o construtor, que é `private clientesService: ClientesService`.

Com a finalização bem-sucedida do `update()`, utilizamos o `router`, que também precisamos injetar em nosso construtor, com `private router: Router`. A princípio, redirecionamos a aplicação para a mesma página, mas isso pode ser dinâmico, dependendo sempre da necessidade em nosso construtor, relacionada à operação concluída. Precisaremos também da injeção `private toastService: ToastService` em nosso construtor.

Para a gravação do novo cliente funcionar, temos que realizar duas alterações. A primeira é na definição do `<form>`, onde devemos inserir `(submit)="submit()"`, que liga nosso método ao processo de submissão do formulário. A segunda, agora sim no botão, é apenas inserir `type="submit"` no código da tag do botão de gravação.

Antes de testarmos nossa aplicação, vamos completar nossa configuração do Firestore na `Console Firebase`. Na página da aplicação, temos um item de menu chamado `Database`, vamos acessá-lo. Ao confirmar o primeiro acesso à base de dados do Firebase, como ainda não temos nenhuma criada, um link para a criação dela será oferecido, então clique nele. Uma janela será exibida, solicitando as regras que devem ser implementadas para o acesso à base. A princípio, vamos liberar o acesso global, selecionando a opção `Iniciar modo de teste`, que não requer autenticação. É possível alterar esta configuração depois e recomendo que realmente o faça. Mais à frente, quando trabalharmos autenticação, retomaremos esse assunto. Confirme o processo clicando o link/botão `Ativar`. A página com a relação de nossas coleções será exibida, mas não temos nada ainda.

Muito bem. Agora podemos testar nossa aplicação para inserirmos um novo cliente. Informe os dados e realize a gravação. Após isso, retorne ao Firebase Console e, em `Databases`, veja o dado já registrado, tal qual podemos ver na figura a seguir.

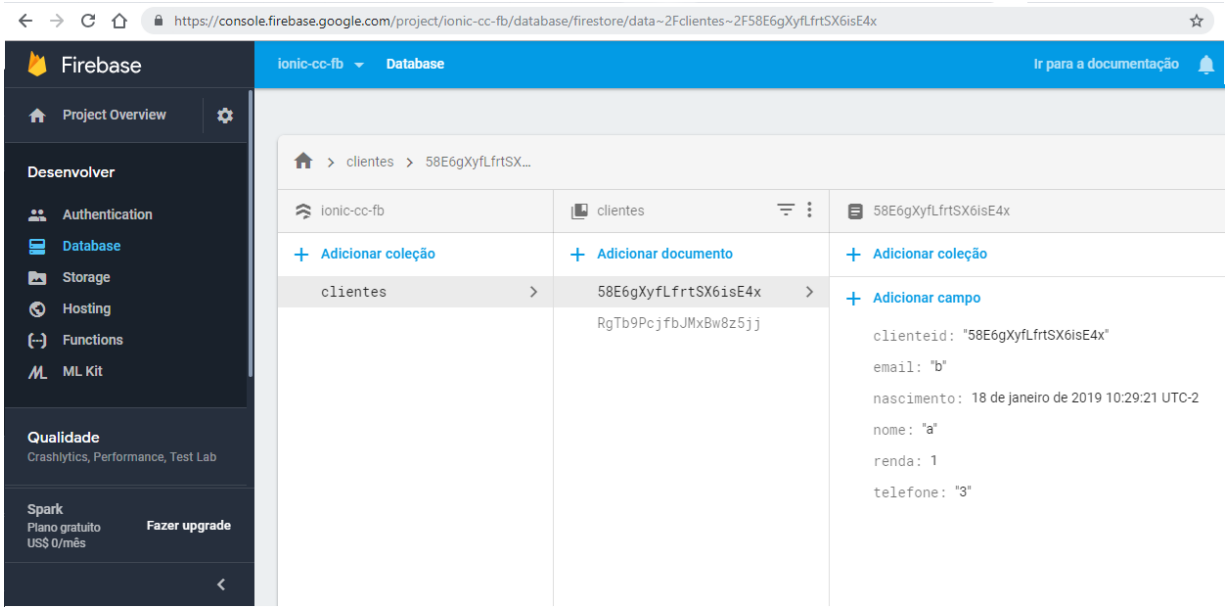


Figura 7.2: Visualização de dados inseridos no Firebase

7.3 Visualização dos clientes registrados

Vamos agora para a implementação da página responsável pela exibição de todos os clientes que já temos persistidos em nossa base de dados remota, no Firestore. Trabalharemos em subseções que apontarão cada arquivo que criaremos ou adaptaremos para esta funcionalidade.

Método getAll() em ClientesService

No capítulo anterior implementamos o método `getAll()` em `ClientesService`, pois precisávamos dos clientes no registro e manipulação dos dados referentes a atendimentos de clientes na aplicação, e funcionou perfeitamente para a tecnologia que adotamos para o momento, que era uma base de dados SQLite. Agora estamos trabalhando com o `Firestore`, precisamos então adaptar este método e podemos vê-lo na sequência. Lembre-se de que ele deverá estar em `clientes.service.ts`.

```
public getAll(): AngularFirestoreCollection<Cliente> {
    return this.firestore.collection('clientes', ref =>
ref.orderBy('nome', 'asc'));
}
```

Observe que retornamos uma coleção genérica de `AngularFirestoreCollection`, que é o tipo de dado retornado pelo método `collection()`. Podemos ver nele a informação de qual a coleção cujos dados queremos e, ainda, o envio de uma `Arrow Function` com critérios para classificação dos dados. Como pode ser visto, é um código bem simples.

Caso você tenha optado por manter seu projeto tal qual tínhamos no capítulo anterior, certamente um erro de compilação ocorrerá com essa nova implementação na página de registro de ordens de serviço, pois a população dos clientes lá esperava um tipo de dado diferente deste que estamos adaptando.

Para contornarmos esta situação neste momento, você pode optar por comentar o código onde há o erro da compilação, até o momento de chegarmos nele, para registrar as adaptações necessárias.

O template para a página de listagem de clientes

Vamos criar uma pasta chamada `listagem` dentro da `/pages/clientes/` e, dentro dela, um arquivo chamado `clientes-listagem.page.html`, com o código que temos na sequência. Temos o `*ngFor` agora com um `pipe` (`|`) para `async`, devido à característica

dos dados que utilizaremos para popular a página. Logo veremos mais sobre ele.

```
<ion-header>
    <ion-toolbar>
      <ion-buttons slot="start">
          <ion-menu-button></ion-menu-button>
      </ion-buttons>
      <ion-title>Clientes</ion-title>
    </ion-toolbar>
</ion-header>

<ion-content padding-left padding-top>
  <ion-list #slidingList>
    <ion-item-sliding *ngFor="let cliente of clientes | async">
      <ion-item href='/menu/(menucontent:clientes-add-
edit/{{cliente.clienteid}})'>
        <ion-label>
          <h2>{{cliente.nome}}</h2>
          <p style="text-align:right">Telefone: {{cliente.telefone}}</p>
        </ion-label>
      </ion-item>

      <ion-item-options side="end">
        <ion-item-option color="danger" (click)="removerCliente(cliente)">
          <ion-button color="danger" no-padding>
              <ion-icon name="trash"></ion-icon>
              Remover
          </ion-button>
        </ion-item-option>
      </ion-item-options>
    </ion-item-sliding>
  </ion-list>

  <ion-fab vertical="top" horizontal="end" slot="fixed" edge="true">
    <ion-fab-button href='/menu/(menucontent:clientes-add-edit/-1)'>
      <ion-icon name="add"></ion-icon>
    </ion-fab-button>
  </ion-fab>
</ion-content>
```

A classe que fornecerá interação para o template de listagem

Na mesma pasta em que criamos o nosso template, vamos agora criar o arquivo `clientes-listagem.page.ts` com o código apresentado na sequência. Observe o comportamento para o método `ionViewWillEnter()`, onde invocamos de maneira encadeada o método `valueChanges()`, que nos retorna um `Observable`, que tipifica o objeto que populará nossa página.

```
import { Component, OnInit, ViewChild } from '@angular/core';
import { Cliente } from 'src/app/models/cliente.model';
import { ClientesService } from 'src/app/services/clientes.service';
import { Observable } from 'rxjs';
import { IonList } from '@ionic/angular';

@Component({
    templateUrl: './clientes-listagem.page.html'
})
export class ClientesListagemPage implements OnInit {

    public clientes: Observable<Cliente[]>;
    @ViewChild('slidingList') slidingList: IonList;

    constructor(
        private clientesService: ClientesService
        ) { }

    ngOnInit() {
    }

    ionViewWillEnter() {
        this.clientes = this.clientesService.getAll().valueChanges();
    }
}
```

O NgModule para a listagem

Já sabemos que precisamos de um terceiro arquivo, o `clientes-listagem.module.ts`. Vamos então criá-lo na mesma pasta dos dois

anteriores.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { IonicModule } from '@ionic/angular';
import { ClientesAddEditPageModule } from '../add-edit/clientes-add-edit.module';
import { ClientesListagemPage } from './clientes-listagem.page';

@NgModule({
    imports: [
      CommonModule,
      FormsModule,
      IonicModule,
      ClientesAddEditPageModule
    ],
    declarations: [
        ClientesListagemPage
    ]
})

export class ClientesListagemPageModule {}
```

Adaptações para testarmos a aplicação

Temos nossa página de listagem já implementada, precisamos agora adaptar nossa aplicação para que, em vez de exibir a página de inserção, exiba a de listagem, pois é a partir da listagem que chegaremos na de inserção.

1. Vamos inserir o código a seguir como um objeto da propriedade `children` , no arquivo `sidemenu-routing.module.ts` .

```
,
{
    path: 'clientes',
    outlet: 'menucontent',
    component: ClientesListagemPage
}
```

2. No `sidemenu.module.ts`, dentro do `imports` do `NgModule`, precisamos inserir a importação para o novo módulo, `ClientesListagemPageModule`, que também deve ser inserido nos `imports` do início do arquivo.

3. Em nosso `sidemenu.page.ts`, precisamos alterar o valor da `url` de clientes para que, ao selecionarmos a opção de clientes na página inicial, o redirecionamento para `clientes` possa ocorrer para a página que estamos concluindo.

4. Novamente no `sidemenu-routing.module.ts`, precisamos configurar a URL para redirecionamento na primeira execução, tal qual podemos ver na sequência.

```
{
    path: '',
    redirectTo: '/menu/(menucontent:clientes)',
}
```

5. Após a gravação do cliente, precisamos redirecionar nosso usuário para a página de listagem. Desta maneira, insira a instrução a seguir abaixo do toast da confirmação de gravação. Lembre-se também de mudar o `redirectTo()` para `clientes`.

```
this.router.navigateByUrl('/menu/(menucontent:clientes)');
```

Execute a aplicação e verifique que a listagem é exibida já com o registro que criamos na implementação anterior.

Vamos a uma curiosidade? Deixe sua página de listagem ativa. Vá ao `Console Firebase` e altere os dados do documento persistido. Volte para sua aplicação e veja que os dados foram atualizados de maneira automática. Mas como isso acontece?

O Firebase trabalha de maneira conectada com nossas aplicações. Você reparou na declaração da propriedade `clientes` anteriormente? O tipo dela é `Observable<Cliente[]>`. Observers são objetos que, por definição de projeto, ficam "escutando" e

esperando acontecimentos que, quando ocorrem, notificam quem os consome.

Na prática, quando atualizamos os dados diretamente no `Console Firebase`, por meio direto ou por diversos consumidores da aplicação, nossa aplicação, em nosso dispositivo, é comunicada da alteração e já atualiza os dados dela. Isso nos garante que os dados estejam sempre atualizados. Tem seus pontos positivos e negativos, o que deve ser sempre avaliado por você e sua equipe.

7.4 Consulta e alteração de um cliente

Vamos trabalhar agora a página de nossa aplicação que exibirá os dados de um cliente selecionado na listagem. Não temos nada de novo em relação à técnica que vínhamos implementando. O que teremos de novidade está ligado diretamente a como nosso serviço será implementado, pois estamos utilizando uma tecnologia nova para nós. Mas veremos que é tranquilo e poderemos discutir alguns pontos.

O método getById() em ClientesService

Tal qual trabalhamos para a listagem, vamos começar as especificidades dessa nova funcionalidade, apresentando a implementação do nosso serviço que realizará a busca do cliente com base no ID do cliente selecionado. Este envio já está definido na rota que implementamos nas seções anteriores, referentes à listagem. Veja o método a seguir.

```
async getById(clienteID: string): Promise<Cliente> {
    try {
      const cliente = await
this.firestore.collection('clientes').doc(clienteID).ref.get();
      if (cliente.exists) {
          const dadosCliente = cliente.data();
```

```
          const nascimento = dadosCliente.nascimento.toDate();
          return  {
              clienteid: dadosCliente.clienteid, nome: dadosCliente.nome,
email: dadosCliente.email,
              telefone: dadosCliente.telefone, renda: dadosCliente.renda,
              nascimento: new Date(nascimento)
          };
      }
    } catch (e) {
      console.log(e);
    }
}
```

Como comentado anteriormente, o Firestore e nossa aplicação, com a implementação que temos, estão ligados. Se retornássemos apenas o documento recuperado, com a instrução `this.firestore.collection('clientes').doc(clienteID)`, teríamos outro tipo de retorno e nosso documento poderia não estar disponível quando precisássemos. Lembre-se de que seria um `Observable` e isso causa uma assincronicidade. Para este momento, não é o que queremos.

Veja o retorno do método anterior, uma `Promise`, que quando começamos a trabalhar com o `SQLite` usamos bastante, o que nos permitirá trabalhar com `async` e `await`.

O foco para este serviço que estamos implementando é termos o objeto recuperado, para que possamos manter a base do que estamos utilizando com formulários nas páginas. Por isso, temos as chamadas `ref.get()` encadeadas ao método `doc()` e nossa invocação principal precedida pelo `await`. Precisamos dela completada e do objeto recuperado, para o retornarmos a quem consumir nosso serviço.

Quando a invocação comentada anteriormente for completada, verificamos se o cliente recuperado realmente existe. Caso positivo, recuperamos seus dados, mapeamos o objeto para um `Cliente` e concluímos com o retorno deste objeto mapeado. Observe que, para

o campo `nascimento`, convertemos o dado que vem do Firestore para uma formatação em data, por meio do `toDate()`. Depois, no envio do objeto, criamos um `Date()` com este valor. Isso se faz necessário pelo fato de o Firestore retornar uma data como `Timestamp`, o que causa erros em nossa aplicação, da maneira como estamos fazendo.

Adaptações para a classe do template

Nós já temos nossa página de inserção de clientes pronta e funcionando e, para o modelo de aplicação que estamos utilizando, precisamos, de início, realizar apenas algumas modificações em nossa classe para o template. Veja a nova implementação para o `ionViewWillEnter()` que está na sequência. Verifique se injetou no construtor o `private route: ActivatedRoute`.

Observe que aplicamos ao código a lógica necessária para verificar se estamos inserindo um novo cliente ou visualizando os dados de um que tenha sido selecionado na listagem. Conseguiu notar a similaridade com a lógica que aplicamos nos capítulos anteriores? Viu que a mudança está praticamente no serviço implementado anteriormente?

```
async ionViewWillEnter() {
    const id = this.route.snapshot.paramMap.get('id');

    if (id !== '-1') {
        this.cliente = await this.clientesService.getById(id);
    } else {
        this.cliente = {clienteid: '', nome: '', email: '', telefone: '',
renda: 0.00, nascimento: new Date()};
        this.modoDeEdicao = true;
    }

    this.clientesForm = this.formBuilder.group({
        clienteid: [this.cliente.clienteid],
        nome: [this.cliente.nome, Validators.required],
        email: [this.cliente.email, Validators.required],
        telefone: [this.cliente.telefone, Validators.required],
```

```
        renda: [this.cliente.renda, Validators.required],
        nascimento: [this.cliente.nascimento.toISOString(),
Validators.required]
    });
}
```

Precisamos garantir uma nova rota em nosso `sidemenu-routing.module.ts`, tal qual podemos ver na sequência, para que a página com os dados do cliente selecionado seja exibida. Podemos também retirar a rota `clientes-add-edit`, pois agora nosso usuário sempre enviará um valor para id.

```
,
{
    path: 'clientes-add-edit/:id',
    outlet: 'menucontent',
    component: ClientesAddEditPage
}
```

Vamos testar nossa aplicação? Selecione um cliente na listagem e veja a página que já conhecemos para a inserção exibindo os dados do cliente que você selecionou.

Possibilitando a alteração dos dados

Como você verificou na execução anterior de nossa aplicação, notando a semelhança com as anteriores, nossos dados são exibidos, mas estão com seus controles desabilitados para alteração e temos, ao final da página, um botão que possibilitará alterar os dados destes clientes. Vamos então realizar as implementações para que este processo todo possa ocorrer. Observe que já implementamos o comportamento para o botão de alteração e para o botão de cancelamento da alteração. Já sabemos que estes métodos devem ser implementados na classe de renderização e nosso template, certo?

```
iniciarEdicao() {
    this.modoDeEdicao = true;
}
```

```
cancelarEdicao() {
    this.clientesForm.setValue(this.cliente);
    this.modoDeEdicao = false;
}
```

Para que possamos testar nossa aplicação, após a implementação anterior, precisamos inserir o código a seguir no componente de nosso botão de alteração que está no template HTML de nossa página. O segundo código é para o botão de cancelamento da edição, que também está em nosso template.

```
(click)="iniciarEdicao()"

(click)="cancelarEdicao()"
```

Agora sim, vamos testar. Selecione um cliente na listagem, inicie a alteração e em seguida grave os novos dados.

A remoção de um cliente para fechamento

Temos nosso CRUD todo estruturado, restando-nos apenas implementar a remoção de um cliente da base de dados. Para isso, temos que implementar o serviço que realize essa operação, ajustar nossa listagem para oferecer ao usuário esta funcionalidade e implementá-la em nossa classe do template. Veja os códigos e neles os comentários.

```
// Método em clientes.service.ts.
async removeById(clienteId: string): Promise<void> {
    return this.firestore.doc(`clientes/${clienteId}`).delete();
}
```

No consumo deste serviço, adotaremos o uso de um alerta para solicitar a confirmação ao usuário para que a remoção possa efetivamente ser executada. Para isso, realizaremos uma implementação em nosso `AlertService`, onde criaremos o método apresentado na sequência.

```
async presentConfirm(header: string, message: string, successFunction)  {
    const alert = await this.alertCtrl.create({
        header,
        message,
        buttons: [
            {
                text: 'Cancel',
                role: 'cancel',
                handler: () => {
                    console.log('Remoção cancelada');
                }
            },
            {
                text: 'Manda brasa',
                handler: () => {
                    successFunction();
                }
            }
        ]
    });
    await alert.present();
}
```

Observou, na assinatura do método, que recebemos menos argumentos em relação ao método que já temos implementado? Para nosso objetivo, que é exibir uma mensagem ao usuário e realizar uma atividade com a confirmação em um dos botões que serão oferecidos, é o que basta. Mas notou o último atributo? Não o tipificamos, mas o nome é suscetível. Se notar mais adiante no código, verá que, caso o usuário confirme a remoção do cliente, essa variável é invocada como um método. Nosso método receberá uma função, mais precisamente, uma `Arrow Function`.

Vamos então implementar o método que consumirá nossos dois métodos recentemente implementados e que estará em nossa classe que renderizará a listagem. Veja-o na sequência. Note a criação da variável `successFunction`, recebendo uma `Arrow Function`, que é a funcionalidade que queremos que seja executada quando o usuário confirmar pela remoção do cliente. Neste código, estamos

consumindo os dois serviços que implementamos, mas lembre-se de que o código de `successFunction` só será executado caso o usuário efetivamente confirmar a remoção.

```
async removerCliente(cliente: Cliente) {
    try {
        const successFunction = () => {
            this.clientesService.removeById(cliente.clienteid);
            this.toastService.presentToast('Cliente removido com sucesso',
3000, 'top');
            this.slidingList.closeSlidingItems();
        };
        await this.alertService.presentConfirm('Remover Cliente',
'Confirma remoção?', successFunction);
    } catch (e) {
        await this.alertService.presentAlert('Falha', 'Remoção não foi
executada', e, ['Ok']);
    }
}
```

Temos nosso CRUD completo! Vamos testar agora a remoção. Na listagem de clientes, busque a exibição do botão para remoção. Ao pressionar o botão, uma popup será exibida, solicitando a confirmação para a operação. Se a operação for confirmada, o cliente será removido e a listagem atualizada.

Atendimento com pesquisa ao cliente

Nós já registramos o atendimento para o cliente no capítulo anterior, fazendo uso do SQLite. Na seleção de clientes, utilizamos um controle, o `<ion-select>`, para exibir os clientes já registrados para escolhermos o do atendimento. Isso funcionou, mas comentamos que não é a melhor solução para selecionar um item quando temos muitas opções para compor a listagem. Veremos agora o uso de uma técnica específica para pesquisa, mais comum em aplicações, sejam móveis, desktop ou web.

Na visão do atendimento, ao lado do controle que exibirá o nome do cliente, teremos um botão que, quando pressionado, abrirá uma

nova visão com a listagem dos clientes recuperados e com um controle onde o usuário poderá digitar o início do nome do cliente, o que permitirá uma filtragem nos dados recuperados para que sejam exibidos apenas os que coincidam com o valor digitado. Esse controle é um `<ion-searchbar>` e veremos aqui o necessário para nossa implementação.

Entretanto, para implementarmos esta técnica, precisaremos adaptar o mecanismo de persistência para o atendimento, que passará a utilizar o Firestore, assim como fizemos para os clientes.

7.5 Visualização dos atendimentos registrados no Firestore

Seguindo nosso padrão de implementação, temos sempre a página com listagem dos dados e, a partir dela, inserimos, consultamos, alteramos e removemos. Vamos para a implementação da página responsável pela exibição de todos os atendimentos que já temos persistidos em nossa base de dados remota, no Firestore - de início, não temos nenhum. Trabalharemos em subseções, que apontarão cada arquivo que criaremos, ou adaptaremos para esta funcionalidade.

Método getAll() em OrdensDeServicoService

No capítulo anterior implementamos o método `getAll()` em `OrdensDeServicoService`, que funcionou perfeitamente para o SQLite, agora, precisamos adaptar para o Firestore. Veja na sequência a nova implementação para o método. Lembre-se de que ele deverá estar em `ordensdeservico.service.ts`.

```
public getAll(): AngularFirestoreCollection<OrdemDeServico> {
    return this.firestore.collection('ordensdeservico', ref =>
ref.orderBy('dataehoraentrada', 'desc'));
}
```

Observe que retornamos uma coleção genérica de `AngularFirestoreCollection`, que é o tipo de dado retornado pelo método `collection()`. Podemos ver nele a informação de qual a coleção cujos dados queremos e ainda, o envio de uma `Arrow Function` com critérios para classificação dos dados. Como pode ser visto, é um código bem simples, semelhante ao que implementamos para Clientes. Precisamos injetar no construtor o código `private firestore: AngularFirestore` e realizar as importações recomendadas pelo VSC, retirando também a injeção para `DatabaseService`. Ainda precisamos retirar os demais métodos do serviço, pois mudamos a estratégia de persistência.

Alteração no template para a listagem

Como trabalharemos com datas persistidas no Firestore, precisamos saber que uma data completa, composta por data e hora, é armazenada nele como `Timestamp`. Isso torna necessária uma alteração em nosso template, que é a inserção à chamada a `toDate()` da propriedade, antes de ela ser enviada ao `pipe`. Veja isso no código a seguir. Caso ocorra um erro na execução apontando para a inexistência do método `toDate()`, ou o resultado não seja o esperado, pode retirá-lo. O mesmo vale para o método `getById()` de `ordensdeservico.service.ts`.

```
<p style="text-align:right">Entrada:
{{ordemDeServico.dataehoraentrada.toDate() | date: 'dd/MM/yyyy'}}</p>
```

Também precisaremos ajustar nosso `*ngFor` para receber os dados de maneira assíncrona. Veja a correção a seguir.

```
<ion-item-sliding *ngFor="let ordemDeServico of ordensDeServico | async">
```

A classe que fornecerá interação para o template de listagem

Precisaremos adaptar o código da classe `OrdensDeServicoListagemPage` do arquivo `ordensdeservico-listagem.page.ts` para que utilize nossa implementação do código anterior, que fez uso do Firestore para recuperação de dados. Tecnicamente é semelhante ao que

implementamos anteriormente para clientes, mas vamos verificar na sequência o código específico para esta seção, com alguns comentários entre eles. A explanação sobre as implementações é a mesma que vimos quando implementamos essa funcionalidade para clientes.

```
// Redefinição da propriedade para o tipo específico para o Firestore
public ordensDeServico: Observable<OrdemDeServico[]>;


// Adaptação do método para recuperar os dados do novo serviço
async ionViewWillEnter() {
    this.ordensDeServico =
this.ordensdeservicoService.getAll().valueChanges();
}
```

Adaptações para testarmos a aplicação

Vamos adaptar nossa aplicação para que exiba o menu de opções que temos e que a página padrão de funcionalidades seja a listagem de atendimentos. Esse é um processo que estamos trabalhando já faz tempo no livro, mas vamos ver aqui novamente detalhes para isso, para facilitar nosso trabalho.

No `sidemenu.module.ts`, precisamos configurar a URL para redirecionamento na primeira execução, tal qual podemos ver na sequência.

```
{
    path: '',
    redirectTo: '/menu/(menucontent:atendimentos)',
}
```

Precisamos também garantir que esta rota esteja registrada no mesmo arquivo, tal qual o código a seguir, além de inserir `OrdensDeServicoListagemPageModule` nos imports de `sidemenu.module.ts`.

```
,
{
    path: 'atendimentos',
```

```
    outlet: 'menucontent',
    component: OrdensDeServicoListagemPage,
}
```

Vamos testar nossa aplicação. Selecione `Atendimentos` no menu e veja que a listagem de atendimentos aparece vazia, o que é natural. Não temos nada registrado ainda no Firestore.

7.6 A pesquisa de clientes para o atendimento

Para que possamos implementar a funcionalidade comentada anteriormente para a realização de pesquisa de clientes para o registro de um atendimento, precisamos implementar todo um componente que oferecerá a interação com o usuário. Seguindo nossa estratégia para explanação, vamos trabalhar com subseções, e cada uma apresentará um artefato para este componente que criaremos.

O template para a página de pesquisa

Em `/src/app/pages` , vamos criar na pasta `clientes` uma pasta chamada `search` . Com esta estrutura preparada, vamos ao nosso arquivo, crie-o com o nome `clientes-search.page.html` e nele implemente o código a seguir.

```
<ion-header>
    <ion-toolbar>
      <ion-buttons slot="start">
      </ion-buttons>
      <ion-title>Pesquisar por CLIENTE</ion-title>
    </ion-toolbar>
</ion-header>

<ion-content no-padding>
    <ion-searchbar placeholder="Nome para filtrar"
(ionInput)="getClientes($event)"></ion-searchbar>
```

```
    <ion-list *ngFor="let cliente of clientes | async"
(click)="clientesClick(cliente)">
        <ion-item>
            <ion-label>
                {{cliente.nome}}
            </ion-label>
        </ion-item>
    </ion-list>
</ion-content>
```

O que temos de novo no código anterior? Conseguiu identificar? Exatamente, o uso de `<ion-searchbar>`. É este componente que permitirá a renderização dos controles relacionados à pesquisa. Temos várias propriedades e eventos que podemos configurar para este componente, mas estamos utilizando apenas o `placeholder`, para um texto ser exibido ao usuário, antes que ele comece a digitação, e o `ionInput`, que será disparado a cada tecla pressionada para compor o nome a ser pesquisado.

Abaixo do `<ion-searchbar>`, temos o já conhecido `<ion-list>`, responsável pela exibição dos clientes já cadastrados, que será filtrada de acordo com o que for sendo informado para pesquisa pelo usuário. Mas como esta filtragem ocorre?

Em primeiro lugar, não há ligação direta entre o controle de pesquisa e a listagem. Não há mágica, precisamos programar. Entretanto, podemos identificar que a listagem, como nos exemplos anteriores, é composta pela varredura em uma variável, chamada `clientes` em nosso caso. Também conseguimos identificar um método chamado `getClientes()`, ligado ao evento `ionInput()`, que receberá um `$event`, que, por sua vez, levará informações do controle `<ion-searchbar>` para nosso código. É neste método que implementaremos o processo captura do que for digitado pelo usuário, para então podermos realizar a seleção dos clientes.

Um novo serviço em ClientesService

Antes de implementarmos a classe que será responsável pela renderização do template anterior, precisamos codificar o método para seleção dos clientes que tenham o nome iniciado pelos valores que serão digitados pelo usuário no controle de pesquisa. Para isso, vamos trabalhar na classe `ClientesService`, que está no arquivo `clientes.service.ts`, na pasta `src/app/services`.

Para implementarmos a solução proposta para este momento, precisamos realizar uma importação de operadores do `RxJS`, para podermos operar na coleção de objetos `Observable` que são recuperados do Firestore. Veja este import na sequência, que deve estar logo no início do arquivo.

```
import { map } from 'rxjs/operators';
```

Com o import codificado, o seguinte código deverá ser implementado ao final da classe. Observe que o método recebe uma string, que conterá o nome, ou o início dele, que deverá ser utilizada como critério de seleção. O retorno do método é o mesmo que definimos para o `getAll()`, que, inclusive, invocamos na primeira instrução do método, populando nossa coleção a ser trabalhada. Caso nenhum valor chegue no argumento `nome`, retornamos a coleção inteira.

```
public getByNome(nome: string): Observable<Cliente[]> {
    const clientes: Observable<Cliente[]> = this.getAll().valueChanges();

    if (!nome) {
        return clientes;
    }

    return clientes.pipe(
        map(result =>
            result.filter(
                cliente =>
cliente.nome.toLowerCase().startsWith(nome.toLowerCase()))
        )
```

```
    );
}
```

E se o argumento `nome` trouxer valores? Aí sim precisamos processá-lo em nossa coleção. E isso é possível por meio do `pipe()`. Já comentamos este método anteriormente, ele processa um valor de entrada e retorna este valor processado. Dentro deste método, estamos invocando outro, o `map()`, que é da biblioteca `RxJS` e que importamos há pouco. Aplicamos a seleção pela invocação ao método `filter()`, utilizando um filtro com base no método `startsWith()` dos caracteres em minúsculos.

A biblioteca `RxJS` é um conjunto de recursos extremamente importante para quem desenvolve aplicações web e, reforçando, aplicações Ionic têm sua base em aplicações web. Essa biblioteca é integrada com aplicações Ionic e Angular e diversos outros frameworks.

A classe TypeScript para nosso template

Vamos agora para a implementação de nossa classe responsável pela renderização de nosso template HTML. Veja o código dela logo na sequência. Crie-a na pasta `pages/clientes/search`, com o nome `clientes-search.page.ts` para o arquivo.

```
import { Component, OnInit, ViewChild } from '@angular/core';
import { Observable } from 'rxjs';
import { Cliente } from 'src/app/models/cliente.model';
import { ClientesService } from 'src/app/services/clientes.service';
import { Location } from '@angular/common';

@Component({
    templateUrl: './clientes-search.page.html',
    styleUrls: ['./clientes-search.page.scss']
})
export class ClientesSearchPage implements OnInit {
    public clientes: Observable<Cliente[]>;

    constructor(
```

```
        private clientesService: ClientesService,
        public location: Location
     ) { }

    ngOnInit() {
    }

    ionViewWillEnter() {
        this.clientes = this.clientesService.getAll().valueChanges();
    }

    getClientes(searchBar) {
        this.clientes =
this.clientesService.getByNome(searchBar.srcElement.value);
    }

    clientesClick(cliente: Cliente) {
        this.location.back();
    }
}
```

No código anterior, observe que no decorator `@Component()` estamos definindo um arquivo para `styleUrls`, onde definiremos um estilo específico para nossa listagem de clientes. Mais à frente veremos este arquivo.

Dentro da classe, temos a definição da propriedade `clientes`, que não é algo novo para nós, pois estamos sempre trabalhando com este tipo de propriedade quando nos referimos a listagens.

No construtor, temos a injeção de `ClientesService`, que possui o serviço de seleção que implementamos recentemente, além do `getAll()`, que invocamos no momento em que a visão recebe o foco, para que todos os clientes possam ser exibidos já na primeira exibição. Isso ocorre no método `ionViewWillEnter()`. Você pode pensar em não fazer uso desta técnica para economizar tráfego desnecessário, pois pode não ser seu objetivo exibir os clientes sem um valor informado para o filtro.

Na sequência, implementamos o método `getClientes()`, recebendo nosso `SearchBar`, onde invocamos o método `getByNome()`, enviando a ele o valor que está digitado no controle. Este método retornará os clientes já com a seleção baseada no valor informado pelo usuário.

O último método, `clientesClick()`, que será invocado quando o usuário escolher um cliente da listagem, a princípio apenas faz uso do objeto `location`, também injetado em nosso construtor, para que a aplicação retorne para a visão anterior à de pesquisa. Logo veremos tudo isso em funcionamento.

A classe para o módulo de pesquisa de clientes

Pois bem, vamos à implementação que se refere à classe onde definimos o módulo. Seguindo nossa convenção, daremos a ela o nome `clientes-search.module.ts` e a criaremos na mesma pasta dos arquivos anteriores. Veja o código na sequência.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { IonicModule } from '@ionic/angular';
import { ClientesSearchPage } from './clientes-search.page';

@NgModule({
    imports: [
      CommonModule,
      FormsModule,
      IonicModule
    ],
    declarations: [
        ClientesSearchPage
    ]
})

export class ClientesSearchPageModule {}
```

O arquivo com definição de estilos para nosso template

Chegamos à etapa final para a implementação de nosso componente de pesquisa, onde vamos criar, junto com os arquivos anteriores, o `clientes-search.page.scss` e nele inseriremos o pequeno código que vemos na sequência. Já conversamos anteriormente sobre este tipo de arquivo e sobre CSS, mas para informação rápida, esta configuração deixará os itens da listagem menores em relação à altura. Quando formos testar nossa aplicação, podemos fazê-lo com este arquivo, para identificarmos a diferença.

```
ion-list {
    margin-bottom: 0% !important;
    margin-top: 0% !important;
}
```

7.7 A criação de um atendimento com acesso à pesquisa por clientes

Ainda não podemos testar nossa aplicação, pois o acesso à visão de pesquisa que criamos anteriormente estará no momento de registro do atendimento, seja na criação de um novo ou alteração de um já existente. Nós temos esta funcionalidade já implementada e a testamos no capítulo anterior, mas o fizemos com outra técnica para seleção do cliente e, ainda, utilizamos o SQLite. Agora, estamos mudando o mecanismo de persistência e o controle para seleção de clientes. Vamos ver estas alterações nas subseções seguintes.

O template para a página do CRUD para atendimento

Como estamos adaptando um código já trabalhado, não trarei para cá todo o template para o CRUD de atendimentos, mas apenas o conteúdo que precisa ser alterado, que, especificamente é a parte onde o cliente deve ser informado. Veja o código a seguir, que está no arquivo `ordensdeservico-add-edit.page.html` .

```
<ion-item lines="none" [disabled]="!modoDeEdicao">
    <ion-label position="floating" class="label-input-title"
style="margin-top: 10%;">Cliente</ion-label>
    <ion-grid no-padding>
        <ion-row no-padding>
            <ion-col size="11" no-padding>
                <ion-input value={{nomeCliente}} disabled="true"
type="text" style="margin-left: -8%;"></ion-input>
            </ion-col>
            <ion-col size="1" no-padding>
                <ion-button (click)="goToClientesSearch()">
                    <ion-icon slot="icon-only" name="search"size="small">
</ion-icon>
                </ion-button>
            </ion-col>
        </ion-row>
    </ion-grid>
</ion-item>
```

Veja que nosso `<ion-input>` está desabilitado, o que desabilita também o `<ion-label>`. É possível você trabalhar com componentes isolados, fora do `<ion-item>`, para impedir esse comportamento, mas deixarei isso a seu cargo, ok? Observe no segundo `<ion-col>` que temos um botão, com uma rota que será configurada mais à frente para nossa página de pesquisa, que criamos agora há pouco. Veja também que no `<ion-input>` estamos realizando a ligação por meio das `{{}}`, que já utilizamos antes.

Para o `<ion-button>`, estamos invocando um método, que nos redirecionará para a página de pesquisa. Aqui é interessante notar que, antes, utilizando o `href`, isso funcionava, mas para esta nova versão, precisei mudar para o código essa navegação. Veja na sequência este método implementado.

```
public goToClientesSearch() {
    this.router.navigateByUrl('/menu/(menucontent:pesquisarclientes)');
}
```

Configuração da rota para a pesquisa de clientes

Vamos verificar no código seguinte o que devemos implementar no arquivo `sidemenu-routing.module.ts`, dentro da coleção `children`, como rota para a pesquisa de clientes, que referenciamos no código anterior para o template como `/menu/(menucontent:pesquisarclientes)`.

```
{
    path: 'pesquisarclientes',
    outlet: 'menucontent',
    component: ClientesSearchPage,
},
```

Algumas mudanças precisarão ser realizadas em nosso projeto. Vamos a elas.

1. Retire o `import { Guid } from 'guid-typescript';` dos imports no arquivo `ordensdeservico-add-edit.page.ts`;
2. Comente, no mesmo arquivo, todo o conteúdo dos métodos `ionViewWillEnter()` e `submit()`;
3. Deixe seu método `ngOnInit()` como o apresentado na sequência;

```
async ngOnInit() {
 this.ordemDeServico = {ordemdeservicoid: '',
     clienteid: '', veiculo: '', dataehoraentrada: new Date() };

 this.osForm = this.formBuilder.group({
     ordemdeservicoid: [this.ordemDeServico.ordemdeservicoid],
     clienteid: [this.ordemDeServico.clienteid, Validators.required],
     veiculo: [this.ordemDeServico.veiculo, Validators.required],
     dataentrada:
[this.ordemDeServico.dataehoraentrada.toISOString(),
Validators.required],
     horaentrada:
[this.ordemDeServico.dataehoraentrada.toISOString(),
Validators.required],
     dataehoraentrada: ['']
 });
```

```
    this.modoDeEdicao = true;
  }
```

4. No arquivo `ordensdeservico-add-edit.page.ts` , insira antes do construtor a declaração `public nomeCliente= 'Pesquise o cliente';` .

Com a criação de nossa nova página, precisamos inserir seu módulo no arquivo `sidemenu.module.ts` . Então, nos imports do `@NgModule` deste arquivo, insira `ClientesSearchPageModule` e garanta que o `OrdensDeServicoAddEditPageModule` também esteja lá. Tudo isso caso você não tenha dado sequência no projeto do capítulo anterior, optando por criar um novo no início deste capítulo.

Também precisamos garantir que a rota a seguir esteja configurada em nosso `sidemenu-routing.module.ts` .

```
{
    path: 'os-add-edit/:id',
    outlet: 'menucontent',
    component: OrdensDeServicoAddEditPage
},
```

Veja na execução da sua aplicação se a visão do atendimento está semelhante à figura a seguir. Lembre-se de que será preciso pressionar `Atendimentos` no menu e depois no botão de inserir novo atendimento para chegar na apresentada pela figura.

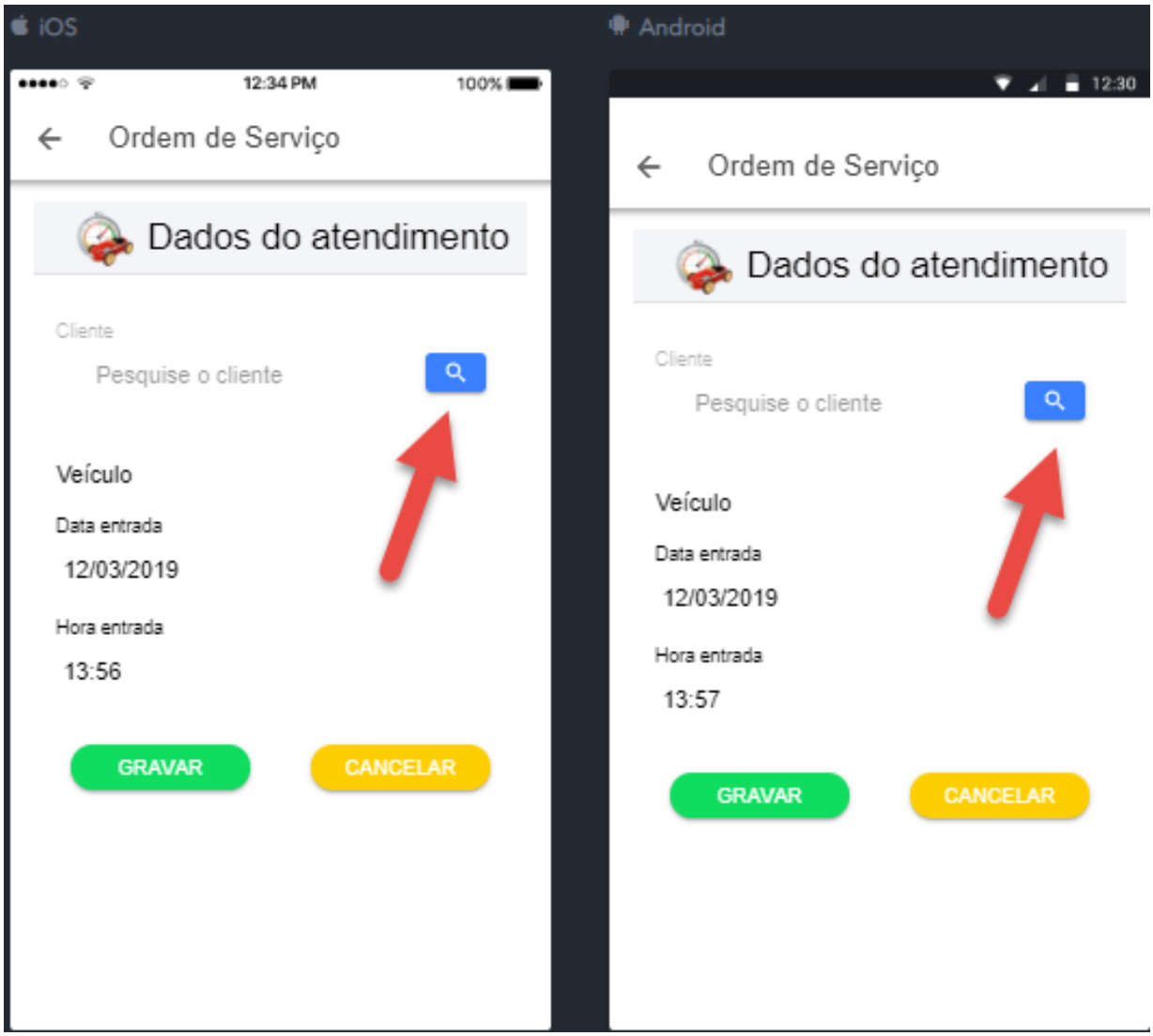


Figura 7.3: Página com a interface para inserção de atendimentos

Viu o destaque na figura anterior para o botão que dispara a pesquisa de clientes? Clicou nele? Se sim, veja na figura a seguir como deve estar sua aplicação.

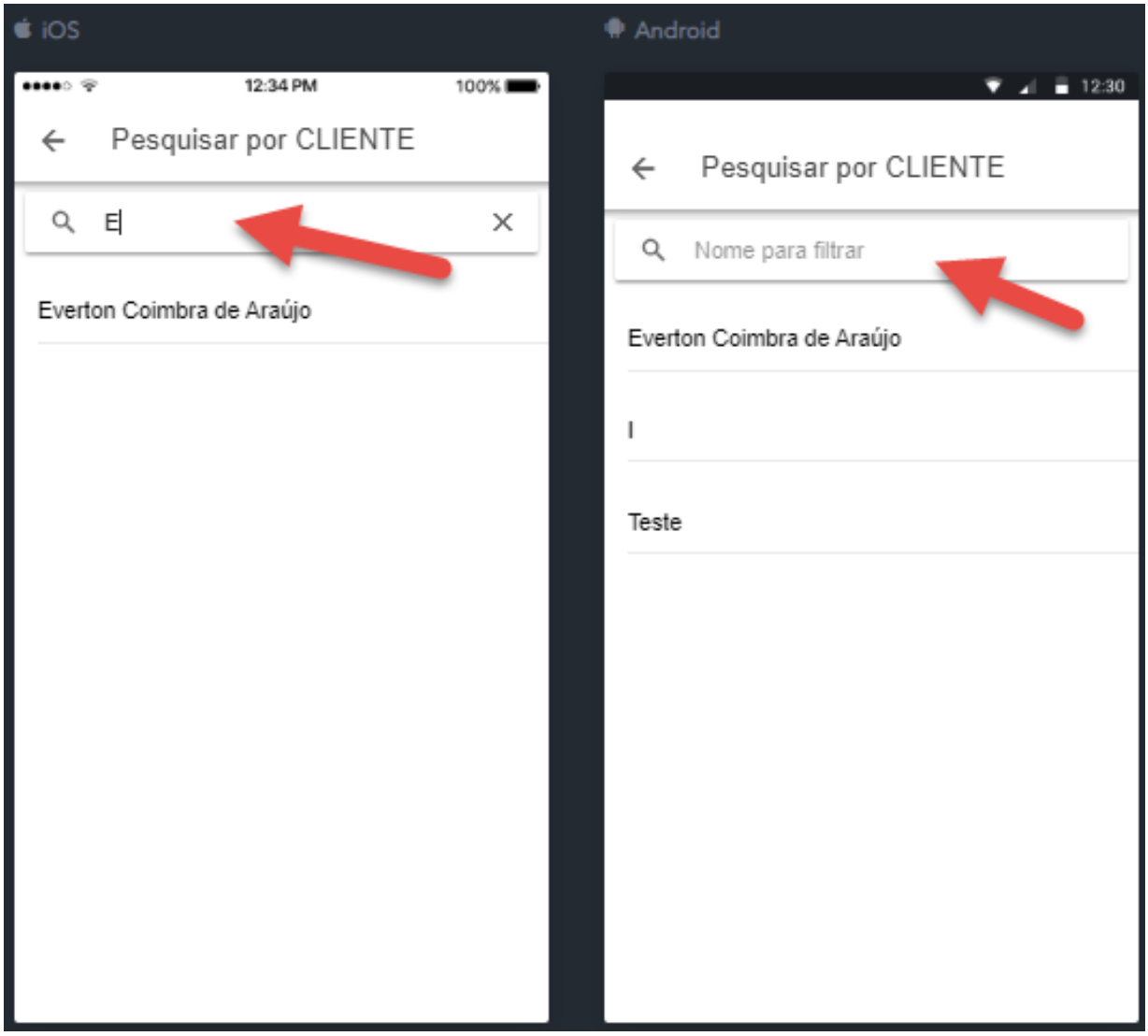


Figura 7.4: Página com a pesquisa de clientes

Notou que no Android não existe valor digitado na pesquisa, o que leva a exibição de todos os clientes registrados? E já no iOS a listagem está exibindo apenas os nomes que começam com E ?

Mas a ideia seria pressionarmos no nome do cliente desejado e que o nome dele fosse exibido em nossa visão de registro de atendimento e isso ainda não está funcionando. Mas vamos implementar esta funcionalidade já na sequência, na próxima seção.

7.8 Eventos na comunicação entre componentes de visão

Em alguns momentos de nossa aplicação teremos a necessidade de comunicação entre uma `tela` e outra (usei o termo `tela` agora para referenciar a visão, nosso template). No caso específico que estamos implementando, no momento em que estivermos pesquisando por um cliente, ao selecionarmos o desejado, precisamos passar para o componente de registro de atendimento este cliente. Vamos fazer isso? É bem simples.

Em primeiro lugar, precisamos injetar um objeto específico para manipulação de eventos, ou seja, precisamos inserir o código a seguir em nosso construtor, no arquivo `ordensdeservico-add-edit.page.ts` .

```
private events: Events
```

O VSC acusará a necessidade de importar `Events` . Acate a sugestão dele, adicionando `Events` ao `import` de `LoadingController` . Seu `import` deverá ser do pacote `@ionic/angular` .

Veja o código a seguir, que deve ser implementado em nossa classe `OrdensDeServicoAddEditPage` , que está no arquivo `ordensdeservico-add-edit.page.ts` . Observe a invocação do método `subscribe()` de `events` , que registrará a assinatura em um evento que aguardará por mensagens. Essas mensagens enviarão um objeto que será recebido e armazenado na variável `data` , que será manipulada na `arrow function` enviada como segundo argumento para `subscribe()` .

O primeiro nome para este método é o nome que daremos ao evento que ficará `escutando` e aguardando por mensagens. É importante que ele seja único. Verifique que recuperamos o `clienteid` recebido e atribuímos ao nosso objeto de formulário e também atribuímos à variável `nomeCliente` o nome recebido, pois ela está ligada ao template e será atualizada quando enviarmos o

cliente selecionado. Após a implementação, precisaremos invocar esse novo método em nosso construtor.

```
private registrarServicoClienteSelecionado() {
    this.events.subscribe('clienteSelecionado', (data) => {
        this.nomeCliente = data.nome;
        this.osForm.controls.clienteid.setValue(data.clienteid);
    });
}
```

Agora precisamos implementar o envio do objeto para a mensagem registrada anteriormente, e faremos isso no momento em que um cliente for selecionado em nossa pesquisa, no `clientes-search.page.ts`. Veja o método a seguir com esta funcionalidade, com comentários na sequência do código.

```
clientesClick(cliente: Cliente) {
    this.events.publish('clienteSelecionado', cliente);
    this.location.back();
}
```

Nós já temos este método implementado e referenciado no template. Falta nele apenas a primeira instrução, que invoca `publish()`, de `events`, que precisamos também injetar em nosso construtor, com a inserção de `private events: Events`.

Podemos testar novamente nossa aplicação. Agora, na visão de pesquisa do cliente, pressione o encontrado e verifique o retorno para a página de registro do atendimento e que, agora, o nome do cliente selecionado aparece no controle.

É preciso uma implementação para finalizarmos esta etapa referente à pesquisa. Nós registramos a assinatura ao evento, mas precisamos agora cancelar esta assinatura, caso contrário, a aplicação estará sempre escutando por algo e, ao sairmos da página de registro de atendimento, não terá mais essa necessidade. Para isso, veja a implementação a seguir, que deve ser implementada em `ordensdeservico-add-edit.page.ts`.

```
public unsubscribeServices() {
    this.events.unsubscribe('clienteSelecionado');
}
```

Verifique que estamos implementando um método, que poderá ser o responsável por cancelar a assinatura aos eventos que tenhamos registrados. Mas quando chamaremos este método? Poderíamos pensar em algum evento do ciclo de vida da visão, mas isso nos daria um certo trabalho no momento, pois precisamos controlar quando registrar e cancelar, já que fazemos uso dele em uma visão chamada por essa assinatura.

Existe o método `ngOnDestroy()`, mas, pela estratégia que estamos utilizando para navegação, esse método não será chamado. Vamos então invocá-lo quando o botão de retornar para a visão anterior for pressionado. Veja a nova implementação para o controle na sequência. Observe a ligação com o evento `click`, em `ordensdeservico-add-edit.page.html`.

```
<ion-back-button defaultHref="/menu/(menucontent:atendimentos)"
(click)="unsubscribeServices()"></ion-back-button>
```

7.9 O registro do atendimento com o cliente selecionado

Já temos os dados para o atendimento devidamente informados pelo usuário. Precisamos agora realizar a gravação deles em nossa base, no Firestore. Já fizemos um processo semelhante quando trabalhamos com clientes, anteriormente no início deste capítulo.

Vamos então a algumas adaptações necessárias para que o processo possa ser realizado agora para atendimentos.

Método update() em OrdensDeServicoService

Toda contextualização sobre a atualização no Firestore já foi realizada anteriormente, assim como a explanação das mudanças necessárias do SQLite para o Firestore. Desta maneira, vamos direto ao novo código para o método `update()`, da classe `OrdensDeServicoService`, que está no arquivo `ordensdeservico.service`. Veja-o na sequência e depois o comento.

```
async update(ordemDeServico: OrdemDeServico): Promise<void> {
    try {
      const osParaGravar: OrdemDeServico = {
        ordemdeservicoid: ordemDeServico.ordemdeservicoid,
        clienteid: ordemDeServico.clienteid,
        veiculo: ordemDeServico.veiculo,
        dataehoraentrada: new Date(ordemDeServico.dataehoraentrada)
      };
      await
this.firestore.doc(`ordensdeservico/${ordemDeServico.ordemdeservicoid}`).s
et(ordemDeServico);
    } catch (e) {
      console.error(e);
    }
}
```

Você observou que estamos criando um objeto `OrdemDeServico`, com dados vindos do argumento recebido, que é um `FormGroup`? Fazemos isso pelo fato de que, em nossa interface com o usuário, nós temos duas propriedades exclusivas para a visão, que são a data e a hora de entrada e, em nossa classe, temos uma única propriedade com estes dois valores. Se persistirmos o objeto recebido diretamente, o Firestore também as persistirá. Teoricamente isso não seria problema, se você tiver estas duas propriedades independentes em seu modelo. Isso pode ficar para uma análise sua. Essa técnica pode lembrar o MVVM, Model-View-

ViewModel e é um tema interessante para se estudar, mas deixo isso à vontade para você.

A invocação do update() no submit() do template

Já temos o serviço que realiza a gravação, quer seja inserção ou atualização, no Firestore. Precisamos agora invocá-lo. Já sabemos, por práticas anteriores, que esse processo é realizado por meio do `submit` do formulário, onde ligamos um evento também chamado, por convenção, de `submit()`. Vamos então implementá-lo, tal qual pode ser verificado no código a seguir.

```
async submit() {
    if (this.osForm.invalid || this.osForm.pending) {
        await this.alertService.presentAlert('Falha', 'Gravação não foi
executada',
            'Verifique os dados informados para o atendimento', ['Ok']);
        return;
    }

    const loading = await this.loadingCtrl.create();
    await loading.present();

    if (this.ordemDeServico.ordemdeservicoid === '') {

this.osForm.controls.ordemdeservicoid.setValue(this.firestoreService.creat
eID());
    }

    this.osForm.controls.dataehoraentrada.setValue(
        this.osForm.controls.dataentrada.value.
            replace(/(\d{4})\-(\d{2})\-(\d{2}).*/, '$1-$2-$3') +
        'T' + this.osForm.controls.horaentrada.value
    );

    await this.ordensDeServicoService.update(this.osForm.value)
        .then(
            () => {
                loading.dismiss().then(() => {
                    this.toastService.presentToast('Gravação bem
```

```
sucedida', 3000, 'top');

this.router.navigateByUrl('/menu/(menucontent:atendimentos)');
                });
            },
                error => {
                console.error(error);
            }
        );
}
```

Verifique, antes da invocação ao `update()`, uma instrução que invoca o método `setValue()` para `dataehoraentrada`. Precisamos mapear a data e hora que temos no `FormGroup` para uma data completa, pois é ela que armazenaremos. O trabalho com datas em JavaScript, em consequência com o TypeScript, não é algo muito trivial. Veja que recuperamos o valor de cada controle e o concatenamos antes de atribuir. Ainda, para recuperarmos a data, que possui o horário quando criamos o controle, fazemos uso de expressão regular, para recuperarmos apenas o que precisamos. Atente para a injeção necessária para o `firestoreService` e `loadingCtrl`.

7.10 A alteração de um atendimento já registrado

Relembrando, nós já temos praticamente toda a implementação necessária para ordens de serviço, pois a trabalhamos no capítulo anterior. O que precisamos é adaptar estas implementações para o uso do Firestore. Faremos isso começando pela refatoração do método `ngOnInit()`, onde retiraremos dele a responsabilidade de criação do FormGroup, para um método específico. Veja este método na sequência, lembrando que nosso arquivo é o `ordensdeservico-add-edit.page.ts`.

```
private createUpdateFormGroup() {
    this.osForm = this.formBuilder.group({
        ordemdeservicoid: [this.ordemDeServico.ordemdeservicoid],
```

```
        clienteid: [this.ordemDeServico.clienteid, Validators.required],
        veiculo: [this.ordemDeServico.veiculo, Validators.required],
        dataentrada: [this.ordemDeServico.dataehoraentrada.toISOString(),
Validators.required],
        horaentrada:
[this.ordemDeServico.dataehoraentrada.toLocaleTimeString('pt-BR'),
Validators.required],
        dataehoraentrada: ['']
    });
}
```

Agora, levaremos para o método `ngOnInit()` a invocação ao método anteriormente criado. Mas vamos trazer aqui todo o método, para relembrar. Veja-o na sequência, e após a listagem trago algumas explicações.

```
async ngOnInit() {
    const id = this.route.snapshot.paramMap.get('id');

    if (id !== '-1') {
        this.ordemDeServico = await
this.ordensDeServicoService.getById(id);
        const cliente = await
this.clientesService.getById(this.ordemDeServico.clienteid);
        this.nomeCliente = cliente.nome;
    } else {
        this.ordemDeServico = {ordemdeservicoid: '', clienteid: '',
veiculo: '', dataehoraentrada: new Date() };
        this.modoDeEdicao = true;
    }

    this.createUpdateFormGroup();
}
```

Por que separamos os métodos? Primeiro, porque para uma boa coesão isso é importante, a separação de responsabilidades. A segunda explicação é que, quando formos cancelar uma operação de alteração, precisaremos atualizar o `FormGroup` para os dados originais do atendimento e, no objeto de atendimento, a variável `this.ordemDeServico` não tem os campos `dataentrada` e `horaentrada`,

que compomos com base no `dataehoraentrada`, como pode ser visto nos códigos anteriores. Observe também, no último código, que invocamos `getById()` de `clientesService`, que precisaremos injetar no construtor. Fazemos isso por precisar do nome do cliente, para que seja exibido na visão.

Para completarmos a implementação da alteração, precisamos codificar o método que será disparado quando o usuário clicar no botão de cancelamento da alteração. Veja este método na sequência.

```
cancelarEdicao() {
    this.createUpdateFormGroup();
    this.modoDeEdicao = false;
}
```

Nós já temos em `ordensdeservico.service.ts` o método `getById()` implementado, mas para uso do SQLite. Agora precisamos adaptar para o Firestore. Veja-o então no código a seguir. Praticamente é o mesmo comportamento que implementamos para clientes. Temos como diferencial o trabalho com a data recuperada, mas é algo facilmente compreendido.

```
async getById(ordemDeServicoId: string): Promise<OrdemDeServico> {
    try {
      const ordemDeServico = await
this.firestore.collection('ordensdeservico').doc(ordemDeServicoId).ref.get
();
      if (ordemDeServico.exists) {
          const dadosOS = await ordemDeServico.data();
          const dataehoradentrada = dadosOS.dataehoraentrada.toDate();

          return  {
            ordemdeservicoid: dadosOS.ordemdeservicoid, clienteid:
dadosOS.clienteid,
            veiculo: dadosOS.veiculo, dataehoraentrada: new
Date(dataehoradentrada)
          };
      }
    } catch (e) {
```

```
      console.log(e);
    }
}
```

7.11 A remoção de um atendimento já registrado

Esta etapa é a final para nosso trabalho deste capítulo, e é bem simples, pois só precisamos mudar nosso serviço, em `ordensdeservico.service.ts` , tal qual podemos ver na sequência.

```
async removeById(ordemDeServicoId: string): Promise<void> {
    return
this.firestore.doc(`ordensdeservico/${ordemDeServicoId}`).delete();
}
```

Também precisamos ajustar nosso método `removerAtendimento()` :

```
async removerAtendimento(ordemdeservico: OrdemDeServico) {
    await
this.ordensdeservicoService.removeById(ordemdeservico.ordemdeservicoid)
        .then( async () => {
            this.ordensDeServico = await
this.ordensdeservicoService.getAll().valueChanges();
            this.toastService.presentToast('Ordem de Serviço removida',
3000, 'top');
            await this.slidingList.closeSlidingItems();
        })
        .catch( async (e) =>  await
this.alertService.presentAlert('Falha', 'Remoção não foi executada', e,
['Ok']));
}
```

Agora podemos executar e testar nossa aplicação criando um novo atendimento, caso nenhum exista. Depois, na listagem, selecione um atendimento já registrado, ele será exibido como consulta, pode alterá-lo e então tentar gravar e cancelar. Depois, seguindo o

procedimento de seleção de um item na listagem de atendimentos, remova um.

Se você quiser testá-la nos dispositivos ou emuladores, tanto utilizando o Cordova como Capacitor, basta utilizar as instruções que já trabalhamos em capítulos anteriores.

Como registro, a aplicação implementada neste capítulo foi testada em emuladores e dispositivos Android e iOS, tanto com Cordova como Capacitor, em ambientes de desenvolvimento Windows e Mac

Conclusão

Terminamos mais este capítulo. Adaptamos nossa aplicação do capítulo anterior, que fazia persistência local no dispositivo para uma persistência distribuída, com o Firestore, mantendo nossos dados na nuvem e exigindo que nosso dispositivo esteja sempre conectado à internet. Podemos unir as duas tecnologias e pensarmos em uma sincronização para trabalho off-line, mas fica o desafio para você essa implementação.

Na nova implementação do registro de atendimentos (ordens de serviços), na seleção do cliente que será atendido, utilizamos uma nova técnica, a de termos uma visão específica para pesquisa por um cliente, com base em valores informados. Não mais apenas selecionamos um cliente de um componente de listagem, como fizemos no capítulo anterior.

Aprendemos também a utilizarmos eventos para promovermos comunicação entre visões, com envios de objetos. Fizemos isso na seleção de um cliente na visão de pesquisa para registro dele no atendimento.

No próximo capítulo trabalharemos com o uso da câmera e álbum de fotografias.

Capítulo 8
Câmera e álbum de fotos no CRUD de clientes e Storage do Firebase

Começamos este novo capítulo com uma implementação de persistência na nuvem realizada. Fizemos isso no capítulo anterior, onde implementamos um CRUD no Firestore, produto do Firebase, do Google. Comentamos sobre a importância de termos uma conexão com a internet para esses serviços, o que é cada vez mais comum.

Em capítulos anteriores realizamos a implementação de CRUD com persistência local também, pois a conexão pode não ser garantida e nosso cliente pode precisar utilizá-la, com vistas a sincronizar os dados locais com a nuvem e também da nuvem com o dispositivo.

Trabalhamos apenas dados textuais, mas sabemos que a existência de imagens para auxiliar na experiência do usuário ou representar pessoas, objetos ou situações é cada vez mais constante. Sabendo disso, não podemos deixar de oferecer funcionalidades para interação da aplicação com a câmera e álbum de fotos dos dispositivos. Veremos isso neste capítulo, que você poderá constatar que é bem interessante.

Iniciaremos o processo interagindo com a câmera, trabalhando com a foto tirada pelo usuário, armazenando-a no dispositivo, de maneira física no ambiente destinado à aplicação, depois, publicaremos a imagem em um `Storage` do `Firebase`, tecnologia e recurso cada vez mais presente em `Cloud`.

Na implementação deste capítulo estou seguindo a que já temos concluída até o capítulo anterior.

8.1 Adaptação de nossa página de clientes

Trabalharemos aqui no livro a implementação com fotos na nossa página de clientes. Permitiremos que o usuário tire uma foto pela câmera, ou selecione uma já armazenada na galeria de fotos do dispositivo.

Para possibilitarmos esta funcionalidade ao nosso usuário, precisamos adaptar nossa interface com ele para a página de CRUD de Clientes, onde teremos uma área para a exibição da imagem e para os botões de interação para ativação da câmera ou seleção no álbum.

Antes de começarmos essa interação, vamos refletir sobre nossa página de clientes. As figuras que vimos no livro demonstram emuladores com telas pequenas e o que já temos como dados para o cliente praticamente ocupa toda a tela. Veremos uma adaptação do `<ion-input>` para ganharmos espaço e podermos ter estes novos controles disponibilizados em nossa página. Também retiraremos a imagem que tínhamos no topo da página. Desta maneira, veja a figura a seguir, com a alteração que implementaremos.


iOS
12:34 PM
100%
Back
Clientes
Nome
Email
Telefone
Renda
0
Nascimento
05/04/2019
Capturar Foto
Gravar
Cancelar

Figura 8.1: Página do CRUD de clientes com controles para a foto do cliente

Nos capítulos anteriores já tínhamos implementado a página com os controles que interagem com o usuário para solicitar os dados de cada cliente, que precisam ser informados para seu registro na aplicação. Agora, vamos redesenhar nossa interface com o usuário. Veja o código inicial para a nova implementação no código a seguir, que é semelhante ao que já tínhamos. Lembre-se de que este código está no arquivo `clientes-add-edit.page.html`, na pasta `src/app/pages/clientes/add-edit`.

```
<ion-header>
    <ion-toolbar>
        <ion-buttons slot="start">
            <ion-back-button defaultHref="/menu/(menucontent:clientes)">
</ion-back-button>
        </ion-buttons>
        <ion-title>Clientes</ion-title>
    </ion-toolbar>
</ion-header>

<ion-content padding>
    <ng-container *ngIf="clientesForm">
        <form [formGroup]="clientesForm" (submit)="submit()">
            <ion-item lines="none" [disabled]="!modoDeEdicao">
                <ion-input formControlName="nome" type="text"
placeholder="Nome"></ion-input>
            </ion-item>

            <ion-item lines="none" [disabled]="!modoDeEdicao">
                <ion-input formControlName="email" type="email"
placeholder="Email"></ion-input>
            </ion-item>

            <ion-item lines="none" [disabled]="!modoDeEdicao">
                <ion-input formControlName="telefone" type="tel"
placeholder="Telefone"></ion-input>
            </ion-item>

            <ion-item lines="none" [disabled]="!modoDeEdicao">
```

```
                <ion-label position="fixed" class="label-input-
title">Renda</ion-label>
                <ion-input formControlName="renda" type="number"></ion-
input>
            </ion-item>

            <ion-item lines="none" [disabled]="!modoDeEdicao">
                <ion-label position="fixed" class="label-input-
title">Nascimento</ion-label>
                <ion-datetime formControlName="nascimento"
displayFormat="DD/MM/YYYY" min="1967" max="2020-10-31"></ion-datetime>
            </ion-item>

            <ion-button *ngIf="!modoDeEdicao" shape="round"
color="primary" expand="block" padding (click)="iniciarEdicao()">
                Alterar dados
            </ion-button>
            <div *ngIf="modoDeEdicao">
                <ion-grid>
                    <ion-row>
                        <ion-col col-6 >
                            <ion-button class="nopadding" shape="round"
color="success" size="small" expand="block" type="submit">
                                Gravar
                            </ion-button>
                        </ion-col>

                        <ion-col col-6>
                            <ion-button class="nopadding"  shape="round"
color="warning" size="small" expand="block" (click)="cancelarEdicao()">
                                Cancelar
                            </ion-button>
                        </ion-col>
                    </ion-row>
                </ion-grid>
            </div>
        </form>
    </ng-container>

</ion-content>
```

Notou o início do template, em que tiramos a imagem de cabeçalho? Conseguiu verificar também que não temos mais todos os `<ion-label>` ? E nos que ficaram, notou que mudamos `position` para `fixed` ? Conseguiu identificar que definimos uma classe de estilo para os `<ion-button>` , chamada `nopadding` ? Tudo para conseguirmos ter todos os controles exibidos sem uso de scroll.

Vamos partir para a implementação dos novos controles, responsáveis pela exibição da fotografia do cliente e interação com a câmera e álbum de fotografias. Utilizaremos o `<ion-card>` para exibir a foto do cliente. Vamos querer que a foto apareça centralizada. Isso nos traz a necessidade de trabalharmos com estilos e já vimos no livro o uso de arquivos específicos para isso, os `scss` . Vamos então criar na pasta `src/app/pages/clientes/add-edit` o arquivo `clientes-add-edit.page.scss` , com o código a seguir.

```
ion-card {
    margin-bottom: 0% !important;
    margin-top: 0% !important;
}

.parent {
    display: flex;
    height: 200px; /* Or whatever */
    background-color: #fff;
}

.child {
    width: 100px;  /* Or whatever */
    height: 100px; /* Or whatever */
    margin: auto;  /* Magic! */
}

.nopadding {
    margin-bottom: 0% !important;
    margin-top: 0% !important;
}
```

No início do código apresentado anteriormente, definimos um estilo específico para o controle `<ion-card>`, para garantirmos o menor espaço possível entre a imagem e os controles acima e abaixo dela. Você pode, e deve, testar a execução sem esta definição, para ver a diferença visual.

As classes `parent` e `child`, definidas também no código anterior, nos auxiliarão na exibição e centralização de nossa imagem. Não entrarei em detalhes específicos sobre `CSS`, por ser um conteúdo extenso, de certa maneira complexo e não ser o escopo de nosso livro. Entretanto, para quem desejar ser um desenvolvedor front-end, esse conhecimento é extremamente necessário.

Em nossa classe `ClientesAddEditPage`, no arquivo `clientes-add-edit.page.ts`, na mesma pasta do template, precisamos informar este novo arquivo criado, para que a renderização o considere. Veja o código a seguir.

```
@Component({
    templateUrl: './clientes-add-edit.page.html',
    styleUrls: ['./clientes-add-edit.page.scss']
})
```

Com o arquivo SCSS criado, implementado e referenciado, vamos ver na sequência o código necessário em nosso template para a exibição da foto e botão com funcionalidades para capturá-la. Este código deverá ser inserido antes do `<ion-button>` para habilitar a alteração dos dados.

```
<ion-card>
    <ion-card-content>
        <div class="parent">
            <ion-img class="child" style="width: 150px; height: 150px;"
src="assets/imgs/icon_clientes.png"></ion-img>
        </div>
    </ion-card-content>
</ion-card>

<div *ngIf="modoDeEdicao">
```

```
    <ion-grid>
        <ion-row>
            <ion-col col-3 >
            </ion-col>

            <ion-col col-6>
                <ion-button class="nopadding"  shape="round"
color="primary" size="small" expand="block">
                    Capturar Foto
                </ion-button>
            </ion-col>

            <ion-col col-3>
            </ion-col>
        </ion-row>
    </ion-grid>
</div>
```

Em relação ao `<ion-card>`, a observação necessária recai sobre o `<div>` e `<ion-img>`, que referenciam as classes CSS implementadas anteriormente. Estamos utilizando uma imagem básica apenas para termos um conteúdo exibido neste momento da implementação.

Após o `<ion-card>`, temos um `<div>` com a exibição condicional. Caso seja exibido, teremos o `<ion-button>` para capturar a foto renderizado. Note que fazemos uso do `<ion-grid>`, para que possamos centralizar o botão horizontalmente abaixo do `<ion-card>`. Também fazemos uso da classe de estilo `nopadding`. Vamos executar e ver como a aplicação está até aqui?

8.2 Interação com a câmera e o álbum de imagens

O processo de seleção de uma imagem para representar a foto do cliente se refere à possibilidade de o usuário selecionar uma imagem já existente na galeria de fotos, o álbum do dispositivo, ou

então, capturar uma imagem diretamente da câmera. Veremos nesta seção estas duas funcionalidades para nosso aplicativo.

Vamos aproveitar que o capítulo é cheio de novidades e trazer mais um controle novo. O `Action Sheet`, comum nas aplicações para dispositivos móveis. Veja-o em execução na figura a seguir.

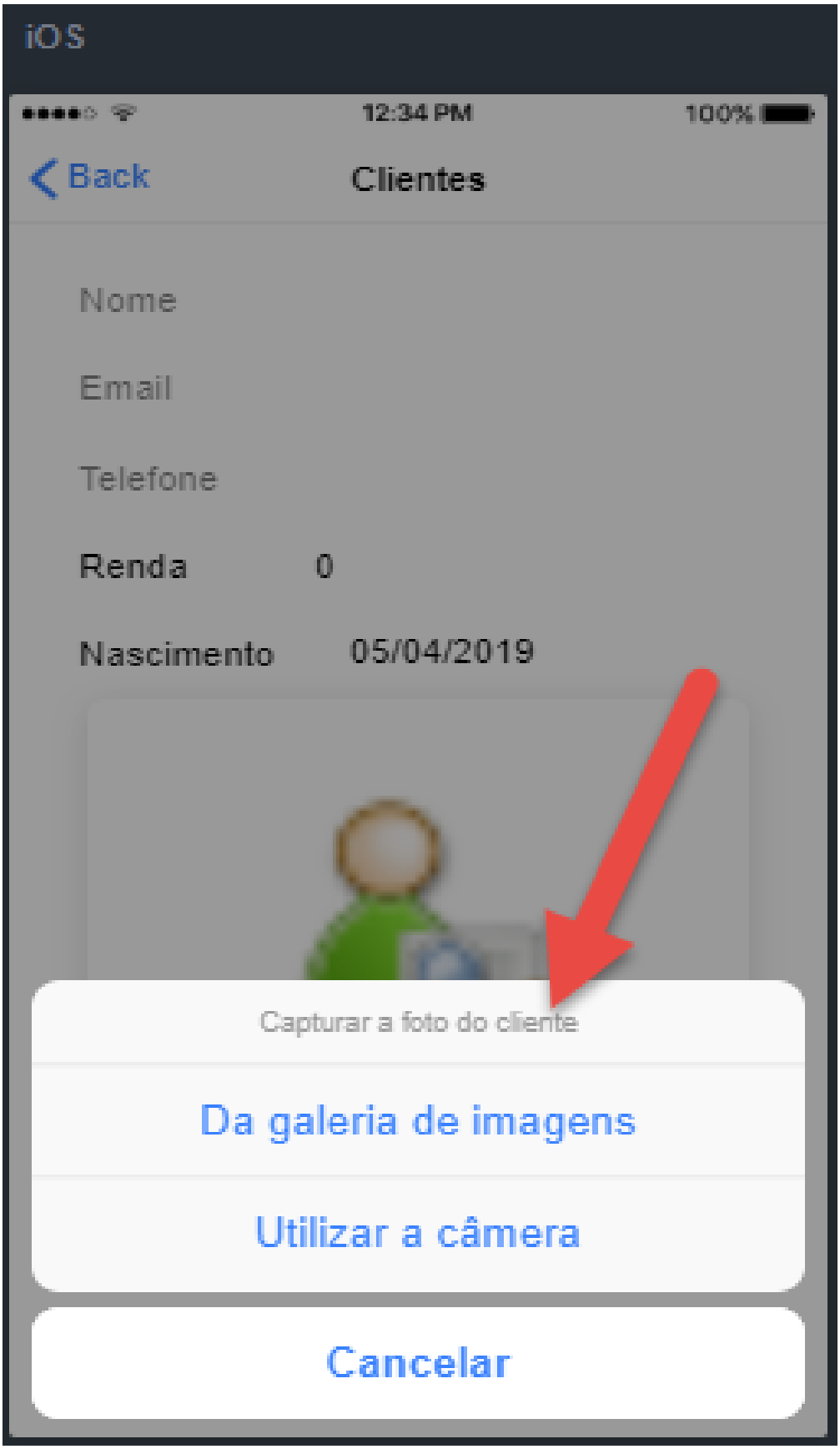

iOS
12:34 PM
100%
Back
Clientes
Nome
Email
Telefone
Renda 0
Nascimento 05/04/2019
Capturar a foto do cliente
Da galeria de imagens
Utilizar a câmera
Cancelar

Figura 8.2: Opções ao usuário com Action Sheet

Nossa ideia é que, ao usuário pressionar o botão para capturar uma imagem, seja oferecida a ele a possibilidade de escolher se a imagem virá da galeria de imagens, ou diretamente da câmera. Precisamos, inicialmente, atribuir ao controle em nosso template a invocação ao método que será responsável por disponibilizar estas opções. Veja o código atualizado a seguir. Observe a ligação com o evento `click`.

```
<ion-button class="nopadding"  shape="round" color="primary" size="small"
expand="block" (click)="capturarFoto()">
```

Com o código para o template adaptado, vamos implementar o método referenciado, o `capturarFoto()`. Veja-o na sequência, lembrando que ele deverá estar em nossa classe `ClientesAddEditPage`, no arquivo `clientes-add-edit.page.ts`, na mesma pasta do template.

```
async capturarFoto() {
    const actionSheet = await this.actionSheetController.create(
    {
        header: 'Capturar a foto do cliente',
        buttons: [{
            text: 'Da galeria de imagens',
            handler: () => {
            }
        },
        {
            text: 'Utilizar a câmera',
            handler: () => {
            }
        },
        {
            text: 'Cancelar',
            role: 'cancel'
        }
        ]
    });
```

```
    await actionSheet.present();
}
```

Observe que a primeira instrução atribui a `actionSheet` um objeto criado por `actionSheetController`. Veja o `this`, que nos informa que esse objeto pertence à nossa classe. Ocorre que ainda não temos o objeto. Precisamos injetá-lo em nossa classe. Para isso, vamos inserir a instrução a seguir no construtor. Será preciso importar `ActionSheetController` na classe.

```
private actionSheetController: ActionSheetController
```

Se você quiser, pode testar essa nova implementação em sua aplicação. Não temos ainda funcionalidades para as opções, pois o `handler()` delas está sem comportamento.

Gostou do visual do `Action Sheet`? Mas vamos implementar a integração com a câmera e álbum de imagens? Você verá que, a princípio, é tudo muito bem simples. Faremos uso de componentes que realizam todo o trabalho pesado para nós. Dessa maneira, na pasta de seu projeto, no terminal, execute as seguintes instruções para instalação do componente e plugin necessário para esta atividade.

```
// Ionic Native Package
npm i @ionic-native/camera@5.8.0


// Cordova Package
ionic cordova plugin add cordova-plugin-camera
```

Com os downloads e instalações realizados, vamos agora codificar. O primeiro ponto é o `import` dos recursos que utilizaremos nas implementações a seguir. Na classe `ClientesAddEditPage`, que está no arquivo `clientes-add-edit.page`, insira logo no início a instrução que se segue.

```
import { Camera, CameraOptions, PictureSourceType } from '@ionic-
native/Camera/ngx';
```

Precisaremos disponibilizar um objeto para nossa classe manipular nossa câmera ou álbum. Já sabemos que podemos fazer isso pelo construtor. Então, insira nele o código a seguir.

```
private camera: Camera
```

Com os códigos anteriores implementados, vamos ao método para capturar a imagem. Veja-o na sequência.

```
obterFoto(sourceType: PictureSourceType) {
    const options: CameraOptions = {
        quality: 10,
        targetHeight: 200,
        targetWidth: 200,
        sourceType,
        saveToPhotoAlbum: false,
        correctOrientation: true,
        destinationType: this.camera.DestinationType.FILE_URI,
        encodingType: this.camera.EncodingType.JPEG,
        mediaType: this.camera.MediaType.PICTURE
      };

    this.camera.getPicture(options).then(caminhoImagem => {
    });
}
```

O método anterior, `obterFoto()`, receberá um objeto que informará ao plugin de onde a imagem será recuperada. Nós vimos na implementação do método `capturarFoto()` que o usuário pode escolher entre a câmera ou álbum de imagens. Usamos este argumento recebido como uma das opções que estamos configurando na variável `options`, do tipo `CameraOptions`. Existem diversas possibilidades de configuração. Não entraremos em detalhes sobre elas aqui. As que utilizamos são semânticas, de fácil abstração. Veja que uma dessas opções refere-se ao argumento recebido pelo método. As três primeiras configurações são importantes para o tamanho físico da imagem. Isso é importante, pois trabalharemos com armazenamento e tráfego de rede.

Precisamos agora invocar este método quando o usuário selecionar o desejo de capturar uma imagem. Vamos realizar esta implementação no método `capturarFoto()`, na `arrow function` para os `handlers`. Veja as implementações a seguir. Optei por trazer todo o código, para ajudar na leitura.

```
async capturarFoto() {
    const actionSheet = await this.actionSheetController.create(
    {
        header: 'Capturar a foto do cliente',
        buttons: [{
            text: 'Da galeria de imagens',
            handler: () => {

this.obterFoto(this.camera.PictureSourceType.PHOTOLIBRARY);
            }
        },
        {
            text: 'Utilizar a câmera',
            handler: () => {
                this.obterFoto(this.camera.PictureSourceType.CAMERA);
            }
        },
        {
            text: 'Cancelar',
            role: 'cancel'
        }
        ]
    });
    await actionSheet.present();
}
```

Precisamos realizar um último ajuste em nosso `app.module.ts` para que possamos concluir esta etapa. Insira a importação a seguir no arquivo.

```
import { Camera } from '@ionic-native/Camera/ngx';
```

Depois, no mesmo arquivo, na lista de Providers, insira `Camera` ao final.

Vamos uma vez mais realizar um teste? Agora vamos ver tanto a câmera quanto o álbum de imagens em execução. Lembre-se de que, para este teste, por utilizarmos recursos do dispositivo, precisaremos realizar o teste em um dispositivo físico, ou emulador.

Para que a câmera possa ser utilizada em dispositivos iOS, a partir do iOS 10, é preciso inserir o código a seguir no `config.xml`, dentro de `<platform name="ios">`.

```
<config-file parent="NSCameraUsageDescription" platform="ios" target="*-Info.plist">
    <string>You can take photos</string>
</config-file>
```

8.3 Armazenando no dispositivo a imagem capturada

Já temos a captura de nossa imagem funcionando. No último teste, vimos que selecionamos a imagem do álbum de imagens ou tiramos uma foto, mas não fizemos, ainda, nada mais que isso. Não temos a imagem exibida, pois ao selecionar a imagem não realizamos nenhum processo em que pudéssemos apontá-la para nosso componente. Temos algumas etapas a vencer e faremos isso aqui nesta seção, por meio de métodos que criaremos e consumiremos, assim como adaptaremos alguns já implementados.

O primeiro método que implementaremos refere-se ao nome que daremos à imagem. Poderíamos pensar em dar a esta imagem o valor que temos para o `id` de cada cliente, pois ele é gerado pelo Firestore e é único. Mas como faríamos para quando estamos inserindo os dados de um cliente novo? Ele ainda não teria este valor. Podemos pensar em algumas possibilidades para esta situação. A primeira seria gerar o `id`, caso ele ainda não exista, mas isso quebraria toda a lógica que já temos para a inserção de um novo cliente, pois o `id` é nosso flag. Poderíamos também

impedir a captura antes de o cliente ter o `id`. Ou seja, para capturar a foto do cliente, ele precisaria antes estar persistido. São duas possibilidades que atingem códigos que já temos implementados e funcionais. As dicas ficam e valem, no mínimo, para uma análise em situações futuras que possam precisar de uma imagem.

Nós tomaremos uma decisão mais simples, mais trivial. Manteremos tudo que já temos implementado e criaremos o arquivo com o nome ligado à data e hora do momento da captura. É mais trivial, mais simples, como dito, mas não atrapalha em nada o que já temos e o que venhamos a ter. É algo independente.

Parece desnecessário, mas vamos criar um método que realize este comportamento, promovendo uma coesão em nossa implementação. Veja este código na sequência. A implementação deve ser feita em nosso arquivo `clientes-add-edit.page.ts`.

```
criarNomeArquivoImagem() {
    const d = new Date(),
        t = d.getTime(),
        novoNomeArquivo = t + '.jpg';
    return novoNomeArquivo;
}
```

Notou a simplicidade do método anterior? Obtemos a data por `Date()` e um valor numérico, que representa a data e hora universal em milissegundos. Veja também que utilizamos uma única vez `const`, para declarar três variáveis. Ao final, retornamos o nome criado.

Vamos a outra funcionalidade cuja necessidade conseguimos antecipar. Toda imagem que obtivermos pela câmera, ou recuperarmos do álbum de imagens, estará persistida fisicamente no dispositivo, em um local que é de controle do sistema operacional, em conjunto com regras específicas para cada aplicativo nele instalado. Normalmente, este local é temporário, assim como o tempo de vida das imagens capturadas. Precisamos gravá-las também no dispositivo, em um local em que a imagem

permanecerá, até que nós a removamos. Veja o código para esta funcionalidade implementado na listagem a seguir.

```
copiarArquivoParaDiretorioLocal(caminho, nomeAtual, novoNomeParaArquivo) {
    this.file.copyFile(caminho, nomeAtual, this.file.dataDirectory,
novoNomeParaArquivo)
        .then(
            success => {
                console.log('Arquivo copiado com sucesso');
        },
            error => {
                this.toastService.presentToast('Erro ao copiar arquivo de
imagem', 3000, 'top');
        });
}
```

Notou a semântica na assinatura do método? O nome dele e o nome para as variáveis relativas aos argumentos recebidos por ele? Esses argumentos são enviados como parâmetros para o método `copyFile()` de um objeto que ainda não temos, chamado `file`. Ainda, na invocação do método `copyFile()`, temos um novo parâmetro, nomeado `dataDirectory`, também de `file`.

Em resumo, para gravarmos um arquivo localmente disponível para a aplicação, originalmente precisamos do caminho onde essa imagem está persistida, o nome que ela tem, normalmente dado pelo dispositivo, de maneira aleatória e o novo nome que será atribuído ao arquivo. Já o método que copiará o arquivo para uma área disponibilizada pelo ambiente para a aplicação, além destes três, precisa do caminho destinado a dados para a aplicação.

A invocação a `copyFile()` retorna um `Promise` e, quando ela for retornada como sucesso (`success`), a princípio, escrevemos algo no console. Caso haja um erro (`error`) neste processo, fazemos uso de nosso já conhecido `ToastService` para informar ao usuário.

Vamos então resolver alguns pontos abordados no código anterior e nas explanações sobre ele. Primeiramente, temos o objeto `file`. Ele se refere a uma injeção que precisamos registrar em nosso

construtor. Entretanto, para que essa injeção possa funcionar, precisamos instalar um componente do Ionic e um plugin do Cordova. Sendo assim, no terminal, na pasta de sua aplicação, digite as instruções a seguir.

```
npm i @ionic-native/file@5.8.0
```

```
ionic cordova plugin add cordova-plugin-file
```

Com a instalação devidamente realizada, precisamos importar estes recursos em nosso arquivo `clientes-add-edit.page.ts` , tal qual o código a seguir.

```
import { File } from '@ionic-native/File/ngx';
```

Agora sim, no construtor de nossa classe, informe o código da sequência.

```
private file: File
```

No método anterior, o `copiarArquivoParaDiretorioLocal()` , teremos o nome e caminho para gravar o arquivo em nosso dispositivo, na área destinada a dados do aplicativo. Ocorre que a URL para o arquivo físico não é a mesma que precisamos para exibir a imagem pelo Ionic. Precisamos converter este caminho para uma URL que possa ser utilizada. Para isso, vamos implementar o seguinte método.

```
caminhoParaImagem(caminhoParaImagem) {
    if (caminhoParaImagem === null) {
        return '';
    } else {
        const caminhoParaArquivoConvertido =
this.webview.convertFileSrc(caminhoParaImagem);
        return caminhoParaArquivoConvertido;
    }
}
```

O código anterior é relativamente simples, não fosse a existência da instrução `this.webview.convertFileSrc(caminhoParaImagem)`, que faz uso de um objeto que ainda não temos. Precisamos deste objeto e, para ele, precisamos de um componente Ionic e um plugin Cordova, que podemos instalar com as instruções a seguir, no Terminal.

```
npm i @ionic-native/ionic-webview@5.8.0
```

```
ionic cordova plugin add cordova-plugin-ionic-webview
```

Repetindo o que fizemos para os componentes anteriores, com a instalação devidamente realizada, precisamos importar esses recursos em nosso arquivo `clientes-add-edit.page.ts`, tal qual o código a seguir.

```
import { WebView } from '@ionic-native/ionic-webview/ngx';
```

E, no construtor de nossa classe, informe o código:

```
private webview: WebView
```

Implementamos alguns métodos até este ponto do livro em relação à persistência física da imagem capturada, mas não os consumimos. Vamos começar pelo `caminhoParaImagem()`, que deve ser consumido pelo método `copiarArquivoParaDiretorioLocal()`, quando o arquivo for copiado com sucesso para nossa área de dados para a aplicação. Para facilitar, veja todo o método novamente.

```
copiarArquivoParaDiretorioLocal(caminho, nomeAtual, novoNomeParaArquivo) {
    this.file.copyFile(caminho, nomeAtual, this.file.dataDirectory,
novoNomeParaArquivo)
        .then(
            success => {
                const caminhoParaArquivo = this.file.dataDirectory +
novoNomeParaArquivo;
                const caminhoParaArquivoConvertido =
this.caminhoParaImagem(caminhoParaArquivo);
                this.imagemSelecionada = caminhoParaArquivoConvertido;
        },
```

```
            error => {
                this.toastService.presentToast('Erro ao copiar arquivo de
imagem', 3000, 'top');
        });
}
```

Verificou que substituímos a exibição de uma mensagem no console, por algumas instruções? A primeira atribui a uma variável o novo nome e caminho para o arquivo que gravamos. A segunda envia este caminho para o método que converterá a URL que utilizaremos para que a imagem seja devidamente exibida. A terceira e última instrução desta nova implementação atribui a uma variável, que ainda não temos, o caminho convertido, para que, por meio de data binding, ele possa ser lido e então ter a imagem exibida na interface do usuário. Vamos declarar essa variável antes de nosso construtor, tal qual o código a seguir:

```
public imagemSelecionada;
```

Agora que temos nossa variável que será ligada com um controle visual, precisamos fazer com que nosso controle, que está com endereço fixo, utilize uma ligação com nossa variável. Veja isso na sequência. Para facilitar, trouxe toda a estrutura do `<ion-card>`, mas só mudamos o controle da imagem.

```
<ion-card>
    <ion-card-content>
        <div class="parent">
            <ion-img class="child" style="width: 150px; height: 150px;"
src={{imagemSelecionada}}></ion-img>
        </div>
    </ion-card-content>
</ion-card>
```

Se tentarmos executar nossa aplicação agora, ao acessarmos a inserção de um cliente, nada aparecerá como imagem do cliente, e isso pode dar uma impressão ruim. Precisamos inicializar esta variável com uma imagem padrão, até que o usuário escolha uma.

Vamos utilizar a mesma imagem que utilizamos antes. Veja isso na implementação a seguir.

```
ngOnInit() {
    this.imagemSelecionada = 'assets/imgs/icon_clientes.png';
}
```

Ainda não podemos testar nossa aplicação para verificarmos essa funcionalidade em execução. Precisamos invocar o método `copiarArquivoParaDiretorioLocal()`, que é o responsável por tudo isso que estamos fazendo. Para facilitar sua leitura, trago a nova implementação completa do método.

```
obterFoto(sourceType: PictureSourceType) {
    const options: CameraOptions = {
        quality: 10,
        targetHeight: 200,
        sourceType,
        targetWidth: 200,
        saveToPhotoAlbum: false,
        correctOrientation: true,
        destinationType: this.camera.DestinationType.FILE_URI,
        encodingType: this.camera.EncodingType.JPEG,
        mediaType: this.camera.MediaType.PICTURE
      };

    this.camera.getPicture(options).then(caminhoImagem => {
        if (this.platform.is('android') && sourceType ===
this.camera.PictureSourceType.PHOTOLIBRARY) {
            this.filePath.resolveNativePath(caminhoImagem)
                .then(filePath => {
                    const caminhoCorrigido = filePath.substr(0,
filePath.lastIndexOf('/') + 1);
                    const nomeUtilizado =
caminhoImagem.substring(caminhoImagem.lastIndexOf('/') + 1,
caminhoImagem.lastIndexOf('?'));
                    this.copiarArquivoParaDiretorioLocal(caminhoCorrigido,
nomeUtilizado, this.criarNomeArquivoImagem());
                });
        } else {
```

```
            const caminhoCorrigido = caminhoImagem.substr(0,
caminhoImagem.lastIndexOf('/') + 1);
            const nomeUtilizado =
caminhoImagem.substr(caminhoImagem.lastIndexOf('/') + 1);
            this.copiarArquivoParaDiretorioLocal(caminhoCorrigido,
nomeUtilizado, this.criarNomeArquivoImagem());
        }
    });
}
```

Verificou a nova implementação? É tudo o que está no bloco da arrow function para `getPicture()`. Observe o comportamento diferenciado para quando a plataforma for a `Android` e a foto vier do álbum de imagens. Uma vez mais estamos utilizando dois objetos que ainda não temos, o `filePath` e o `platform`. Vamos antes entender o código comum. São três as instruções: obter o caminho correto (`correctPath`) e o nome atual do arquivo (`currentName`). Utilizamos estes dados para invocar o método `copiarArquivoParaDiretorioLocal()`.

Voltando ao caso específico do `Android`, para obtermos os dados anteriores, precisamos antes invocar o método `resolveNativePath()`, do objeto `filePath`, que precisamos ter em nossa classe. Esse método, como o nome busca dizer, resolve o caminho nativo do arquivo para uma URL comum. No terminal, execute as instruções a seguir.

```
npm i @ionic-native/file-path@5.8.0
```

```
ionic cordova plugin add cordova-plugin-filepath
```

Para termos o componente disponível em nossa classe, precisamos importar estes recursos em nosso arquivo `clientes-add-edit.page.ts`, tal qual o código a seguir.

```
import { FilePath } from '@ionic-native/file-path/ngx';
```

No construtor de nossa classe, informe o código da sequência.

```
private platform: Platform,
private filePath: FilePath
```

Agora temos o endereço de nossa imagem do cliente ligado ao nosso modelo e precisamos ajustar isso. Nosso primeiro passo é em nossa interface `Cliente`, localizada em `/src/app/models/cliente.model.ts`, inserir nossa nova propriedade, ao final do código, após `nascimento`, como no código a seguir.

```
foto: string;
```

Temos agora alguns pontos para associarmos nossa nova propriedade. O primeiro se dará no método `submit()`, antes da invocação ao `await this.clientesService.update(this.clientesForm.value)`, inserindo a instrução a seguir, que não é novidade para nós. Apenas atribuímos a um controle de nosso formulário um valor, que neste caso é a imagem que temos como selecionada para o cliente atual.

```
this.clientesForm.controls.foto.setValue(this.imagemSelecionada);
```

Mas e quando estivermos acessando um cliente já cadastrado, que pode ter uma foto já informada? Precisamos dela exibida. Vamos fazer isso na declaração do objeto `cliente`, de nossa classe. Veja o código a seguir, que faz parte do `ionViewWillEnter()`. Veja a nova implementação tanto para o `if`, como para o `else`, que tratam a recuperação de um cliente já existente, quanto a de um novo, respectivamente.

```
if (id !== '-1') {
    this.cliente = await this.clientesService.getById(id);
    this.imagemSelecionada = this.cliente.foto;
} else {
    this.cliente = {clienteid: '', nome: '', email: '', telefone: '',
renda: 0.00, nascimento: new Date(),
        foto: this.imagemSelecionada };
    this.modoDeEdicao = true;
}
```

Para concluirmos esta etapa, no mesmo método da listagem anterior, onde declaramos nosso objeto responsável pelo formulário, precisamos também adotar a nova propriedade `foto`. Para facilitar, trago na sequência a declaração completa do objeto.

```
this.clientesForm = this.formBuilder.group({
    clienteid: [this.cliente.clienteid],
    nome: [this.cliente.nome, Validators.required],
    email: [this.cliente.email, Validators.required],
    telefone: [this.cliente.telefone, Validators.required],
    renda: [this.cliente.renda, Validators.required],
    nascimento: [this.cliente.nascimento.toISOString(),
Validators.required],
    foto: [this.cliente.foto]
});
```

Para podermos testar nossa aplicação, precisamos alterar nosso método `getById()` de `clientes.service.ts`, para que o `return` inclua `, foto: dadosCliente.foto` ao final da instrução que lá temos. Também precisamos, no `app.module.ts`, no `providers` do `@NgModule`, inserir ao final `File, WebView, FilePath`. Para garantir que os imports para estes providers estejam devidamente importados, veja se está de acordo com o código a seguir.

```
import { WebView } from '@ionic-native/ionic-webview/ngx';
import { FilePath } from '@ionic-native/file-path/ngx';
import { File } from '@ionic-native/File/ngx';
```

Com toda esta grande etapa vencida, podemos testar nossa aplicação. Recupere uma imagem do álbum de fotos e da câmera. Para o Android isso pode ser feito no emulador. Para o iPhone, precisamos de um dispositivo.

8.4 Visualização das imagens dos clientes na listagem

Já temos a inserção funcionando de maneira satisfatória, em relação ao objetivo que tínhamos. Agora, precisamos adaptar nossa listagem, para que a foto de cada cliente seja exibida também na listagem, e não apenas no momento da gravação. Para isso, vamos adaptar nosso `<ion-item>` para o código a seguir. Algumas adaptações também foram feitas buscando um melhor visual. Este código está em `clientes-listagem.page.html` .

```
<ion-item href='/menu/(menucontent:clientes-add-
edit/{{cliente.clienteid}})'>
    <ion-thumbnail slot="start">
        <ion-img [src]="cliente.foto"></ion-img>
    </ion-thumbnail>
    <ion-grid>
      <ion-row>
        <ion-label><p style="font-size: larger; font-weight: bold;">
{{cliente.nome}}</p></ion-label>
      </ion-row>
      <ion-row>
        <ion-col col-12>
          <ion-label><p style="font-weight: 400; text-align:
right">Telefone: {{cliente.telefone}}</p></ion-label>
        </ion-col>
      </ion-row>
    </ion-grid>
</ion-item>
```

Observou o uso do `<ion-thumbnail>` para nossa imagem? Verificou a ligação de `[src]` com `cliente.foto` ? Para os dados de cada cliente, estamos utilizando um `<ion-grid>` , componente já conhecido por nós. Você pode uma vez mais testar sua aplicação, e verificará que a imagem que selecionamos para o cliente, em nosso último teste, aparece na listagem, ao lado dos dados do cliente que inserimos.

8.5 Remoção da imagem física do dispositivo

Mas e na remoção do cliente? A imagem ficará ainda no dispositivo. Não queremos isso, e nem nosso usuário, correto? Vamos então adaptar um pouco nosso código do método `removerCliente()`. Para relembrar, trago toda a implementação na sequência, para ajudar na compreensão. Ele é no TypeScript da listagem e lembre-se de realizar a injeção para `file`, no construtor.

```
async removerCliente(cliente: Cliente) {
    try {
        const successFunction = async () => {
            this.clientesService.removeById(cliente.clienteid);

            const caminho = cliente.foto.substr(0,
cliente.foto.lastIndexOf('/') + 1);
            const nomeArquivo =
cliente.foto.substr(cliente.foto.lastIndexOf('/') + 1,
(cliente.foto.length - caminho.length));
            const caminhoParaArquivo = this.file.dataDirectory +
nomeArquivo;

            try {
                await this.file.removeFile(this.file.dataDirectory ,
nomeArquivo);
                this.toastService.presentToast('Cliente removido com
sucesso', 3000, 'top');
                this.slidingList.closeSlidingItems();
            } catch (e) {
                await this.alertService.presentAlert('Falha', 'Remoção do
arquivo não foi executada', e, ['Ok']);
            }
        };
        await this.alertService.presentConfirm('Remover Cliente',
'Confirma remoção?', successFunction);
    } catch (e) {
        await this.alertService.presentAlert('Falha', 'Remoção não foi
executada', e, ['Ok']);
    }
}
```

Notou, na declaração de `successFunction`, que após a invocação do `removeById()` declaramos três variáveis? Elas são necessárias para termos o nome do arquivo completo, com base em um endereço físico, pois convertemos esse endereço para um valor que fosse mapeado facilmente para ser renderizado. Poderíamos pensar em um método específico para isso, ou então trabalhar o mesmo método de conversão, o `caminhoParaImagem()`, que fizemos na classe de inserção e edição de clientes, para atender às duas situações. Ainda, é possível usarmos o `LoadingController`, caso você julgue necessário para o usuário não ter a impressão de congelamento da aplicação.

Imagine que o usuário selecionou uma imagem, na inserção ou alteração de um cliente e, ao invés de gravar, ele cancela o processo. O que acontece com a imagem capturada na inserção? Ela precisa ser eliminada. E se ele cancela em uma alteração? É preciso manter a anterior e eliminar a nova. Fica o desafio.

8.6 Vamos manter a foto também na nuvem

Sabemos que o espaço físico disponível em um dispositivo móvel, dependendo do modelo, fabricante, plataforma, é algo caro e precisamos otimizar este uso. Vimos no capítulo anterior o uso do Firestore para persistirmos nossos dados na nuvem. Por que não termos, então, nossa foto também na nuvem? Utilizaremos mais um produto do Firebase do Google, o `Storage`.

Precisamos preparar este serviço para nossa aplicação no console do Firebase. Desta maneira, com seu acesso realizado, clique na opção chamada `Storage`, que normalmente está em um menu ao lado esquerdo de sua tela. Opto por não apresentar uma imagem da console do Firebase por ser uma aplicação web e poder mudar facilmente.

Ao acessar pela primeira vez a opção Storage, algumas opções orientativas serão oferecidas, recomendo que você as leia. Podemos clicar no botão de `Primeiros passos` para começarmos a configuração deste serviço para nosso aplicativo.

Será apresentado a você um script que registrará as regras para acesso de leitura e escrita para este serviço. Como estamos em fase de desenvolvimento e testes e ainda não temos autenticação e muito menos autorização, vamos alterar a instrução responsável por isso no código. De início, clique na opção de confirmação.

Com este processo concluído, você verá uma mensagem semelhante a `As regras padrão de segurança exigem que os usuários se autentiquem`. É o que comentamos no parágrafo anterior. Clique na opção `Regras` no topo da página do console. Substitua a instrução de permissão (`allow`) pela que está a seguir, que permite acesso a todos que tenham as credenciais da aplicação.

```
allow read, write: if true == true;
```

Ao informar o código anterior, uma mensagem de opção de publicação da alteração, ou descarte dela, é imediatamente apresentada. Confirme a publicação. Podemos voltar à relação de arquivos, clicando na opção `Arquivos`.

Lembre-se de que precisamos configurar nosso `credentials.js` para que nossa aplicação possa fazer uso do Storage, o que fizemos no capítulo 7.

Vamos agora trabalhar nas novidades. Teremos alterações de código e novas implementações. Para a técnica que adotaremos, que será enviar a imagem apenas na gravação do cliente, precisaremos ter uma variável em nossa classe, que seja privada e que tenha a URI do arquivo da imagem obtida. Para isso, no início da classe, declare da seguinte maneira no `clientes-add-edit.page.ts`.

```
private uriArquivoImagem;
```

Trabalharemos em nosso método `obterFoto()` já existente na classe `ClientesAddEditPage`, do arquivo `clientes-add-edit.page.ts`, inserindo logo após sua assinatura a declaração a seguir para três variáveis que utilizaremos para essa nova implementação.

```
let caminhoCorrigido, nomeUtilizado;
```

Na sequência, adaptaremos a invocação para o método `getPicture()`, que está no mesmo método que estamos trabalhando. Para facilitar, trago todo o código, para que você possa comparar com o que já temos e ser melhor para a explicação que estará após a listagem.

```
this.camera.getPicture(options).then(async (caminhoImagem) => {
    this.loading = await this.loadingCtrl.create({
      message: 'Please wait...'
    });
    await this.loading.present();

    if (this.platform.is('android') && sourceType ===
this.camera.PictureSourceType.PHOTOLIBRARY) {
        const caminhoArquivo = await
this.filePath.resolveNativePath(caminhoImagem);
        caminhoCorrigido = caminhoArquivo.substr(0,
caminhoArquivo.lastIndexOf('/') + 1);
        nomeUtilizado =
caminhoImagem.substring(caminhoImagem.lastIndexOf('/') + 1,
caminhoImagem.lastIndexOf('?'));
    } else {
        caminhoCorrigido = caminhoImagem.substr(0,
caminhoImagem.lastIndexOf('/') + 1);
        nomeUtilizado =
caminhoImagem.substr(caminhoImagem.lastIndexOf('/') + 1);
    }

    this.uriArquivoImagem = caminhoCorrigido + nomeUtilizado;
    this.imagemSelecionada =
this.caminhoParaImagem(this.uriArquivoImagem);
});
```

Verifique no código anterior, com o que já temos implementado, que alteramos a invocação para `resolveNativePath()` para `await`, em vez de `Promise`. Poderíamos manter, mas quis trabalhar as atribuições para as variáveis envolvidas de maneira diferente, justificando as declarações que fizemos anteriormente. Veja que utilizamos as variáveis após a avaliação condicional do `if..else` para a atribuição da URI do arquivo recuperado e também para converter essa URI de maneira que possa ser ligada ao nosso componente de imagem. Com esta implementação já temos o necessário para começarmos a pensar no envio de nosso arquivo de imagem obtido para o `Storage` do Firebase. Precisamos realizar a declaração a seguir antes do construtor da classe.

```
private loading;
```

Nosso método anterior apenas recupera a imagem e a exibe na visão do usuário. Nada diferente do que tínhamos até a etapa anterior. Apenas não estamos copiando o arquivo recuperado para a área da aplicação, pois não usaremos este arquivo. Precisamos enviá-lo para o `Storage`, que aceita receber dados do tipo `File` e `Blob`. Nós transformaremos nosso arquivo recuperado para `Blob` e então o enviaremos.

Ocorre que a metodologia que estamos adotando realizará o upload no momento de gravar o cliente no `FireStore`. Desta maneira, precisaremos criar alguns métodos em nossa classe `ClientesService`, no arquivo `clientes.service.ts`. Veja o método a seguir, com comentários após ele. Entretanto, para que o método possa ser compilado, precisamos injetar `private file: File` no construtor e importar `import { File } from '@ionic-native/File/ngx';` no início do arquivo. Também precisamos inserir, após a última importação, a `import * as firebase from 'firebase';`.

```
criarBlobDeArquivoDeImagem(caminhoArquivo) {
    return new Promise((resolve, reject) => {
      this.file
        .resolveLocalFilesystemUrl(caminhoArquivo)
        .then(dadosDoArquivo => {
```

```
            const { name, nativeURL } = dadosDoArquivo;
            const path = nativeURL.substring(0,
nativeURL.lastIndexOf('/'));
            return this.file.readAsArrayBuffer(path, name);
        })
        .then(buffer => {
            const blobDaImagem = new Blob([buffer], {
                type: 'image/jpeg'
            });
            resolve(blobDaImagem);
        })
        .catch(e => reject(e));
    });
}
```

Verifique que o método todo se baseia no retorno de uma `Promise`, que inicialmente busca resolver o caminho recebido para o arquivo. As variáveis `name` e `nativeURL` são declaradas como string, recebendo os dados de `dadosDoArquivo`, que é do tipo `Entry`, que tem propriedades de mesmo nome das variáveis declaradas. Com esses dados recuperados podemos ler o arquivo como se fosse um buffer para uma matriz, que é enviado para o segundo `then()`, onde o `Blob` é criado e retornado.

Falta agora a implementação do método que envia o Blob (arquivo) para o Storage. Vamos então implementá-lo, tal qual código apresentado na sequência, também na classe `ClientesService`.

```
async upload(imagem, nomeImagem) {
    if (imagem) {
        try {
            nomeImagem = 'fotosClientes/' + nomeImagem;
            await firebase.storage().ref().child(nomeImagem).put(imagem);
        } catch (e) {
            console.log(e);
        }
    }
}
```

Observou a atribuição da variável `nomeImagem`, com uma pasta antecedendo o nome do arquivo para a imagem recuperada? Com o nome definido, invocamos o método `put()` do `storage()` que, por sua vez, retorna uma referência para o caminho definido. É essa instrução que envia a imagem para o `Storage` do Firebase.

Vamos à implementação de um método na classe `ClientesService`, que é relacionado ao nome do arquivo que será armazenado no `Storage`, o `criarNomeArquivoImagem()`. Se lembrar da nossa seção anterior, criamos este método para dar o nome ao arquivo físico que criamos. Ele foi criado na classe `ClientesAddEditPage`, mas precisamos dele também em `ClientesService`. Analisando nossas classes, temos na primeira a injeção da segunda. Então, podemos trazer esse método para `ClientesService` e, se precisarmos dele em `ClientesAddEditPage`, fica mais fácil o uso.

Vamos adaptar nosso método `update()`, da classe `ClientesService`, para o código apresentado na sequência. São poucas mudanças, mas preferi trazer o código completo:

```
async update(cliente: Cliente): Promise<void> {
    try {
      cliente.nascimento = new Date(cliente.nascimento);
      await
this.firestore.doc(`clientes/${cliente.clienteid}`).set(cliente);

      cliente.foto = cliente.foto.replace('http://localhost/', 'file:///');

      const blobDaImagem = await
this.criarBlobDeArquivoDeImagem(cliente.foto);
      await this.upload(blobDaImagem, this.criarNomeArquivoImagem());
    } catch (e) {
      console.error(e);
    }
}
```

Notou que as mudanças promovidas no código anterior estão na chamada aos dois métodos que implementamos anteriormente e ao que trouxemos para a nossa classe `ClientesService`? Precisamos

também implementar o método a seguir, que estamos invocando no método anterior, mas já foi pedido anteriormente para trazer para esta classe. Verificou também um *workaround* na correção do caminho do arquivo de `http` para `file`? Isso se fez necessário após atualizações do Ionic.

```
criarNomeArquivoImagem() {
    const d = new Date(),
        t = d.getTime(),
        novoNomeArquivo = t + '.jpg';
    return novoNomeArquivo;
}
```

Falta um ajuste: precisamos retirar o código a seguir da invocação ao `getPicture()`, em `obterFoto()` da classe `ClientesAddEditPage`, pois nosso *loading control* está agora na gravação e não na escolha do arquivo.

```
this.loading = await this.loadingCtrl.create({
  message: 'Please wait...'
});
await this.loading.present();
```

Execute a aplicação, insira ou altere um cliente, capture uma imagem e, após ela ser exibida, grave os dados do cliente. Acesse o `Storage` na console do Firebase e veja lá a nova pasta criada, com o arquivo gravado nela.

8.7 Recuperar a foto da nuvem para exibição

Temos os dados de nossos clientes na nuvem, também temos as fotos deles lá. Entretanto, em nossa listagem de clientes, estamos apresentando a foto pelo arquivo da recuperação dela, que já comentamos ter persistência temporária, mas nós queremos que os arquivos sejam permanentes, até que o usuário os remova, e que fiquem na nuvem. Vamos começar a trabalhar nisso agora então.

No processo de inserção de um cliente ou alteração dos dados de um já existente, temos sempre a possibilidade do usuário enviar ou não uma foto. Não deixamos como algo obrigatório. Então, só precisamos realizar o upload se uma foto foi selecionada. Ainda, temos a situação de já existir uma foto no cadastro do cliente e o usuário resolver trocá-la. Precisamos prever e controlar estas situações, que estão todas centralizadas na captura ou não de uma foto. Ou seja, depende dos dados da visão. Veja o novo código para a classe `Cliente`, que está no arquivo `cliente.model.ts`, em nossa pasta `models`.

```
export interface Cliente {
    clienteid: string;
    nome: string;
    email: string;
    telefone: string;
    renda: number;
    nascimento: Date;
    uriFotoCapturada: string;
    nomeFotoEnviada: string;
    uriFotoParaExibir: string;
}
```

Notou na listagem anterior a retirada da propriedade `foto` e a inserção de três novas? A primeira, `uriFotoCapturada`, será nosso flag para verificar se houve ou não captura de uma foto por parte do usuário. A segunda, `nomeFotoEnviada`, apenas para termos o nome dado à foto, caso precisemos dele no futuro. A terceira, `uriFotoParaExibir`, terá o que realmente precisamos, que é o endereço de nossa foto que deverá ser exibida.

Agora precisamos alterar o componente `<ion-img>` de nosso arquivo `clientes-listagem.page.html` para fazer a ligação com nossa nova propriedade. Veja o novo código na sequência.

```
<ion-img [src]="cliente.uriFotoParaExibir"></ion-img>
```

Precisamos alterar o método `getById()` de nossa classe `ClientesService` do arquivo `clientes.service.ts`, para mapear as

novas propriedades. Veja a alteração no código a seguir.

```
return  {
  clienteid: dadosCliente.clienteid, nome: dadosCliente.nome, email:
dadosCliente.email,
  telefone: dadosCliente.telefone, renda: dadosCliente.renda,
  nascimento: new Date(nascimento), uriFotoCapturada: dadosCliente.
uriFotoCapturada,
  nomeFotoEnviada: dadosCliente.nomeFotoEnviada, uriFotoParaExibir:
dadosCliente.uriFotoParaExibir
};
```

Para nossa próxima alteração, precisamos pensar na situação do usuário acessar a visão de inserção/consulta/alteração de cliente. Vamos então para o método `ionViewWillEnter()` de nossa classe `ClientesAddEditPage`. Trago-o na íntegra na sequência para facilitar a visualização das alterações.

```
async ionViewWillEnter() {
    const id = this.route.snapshot.paramMap.get('id');

    if (id !== '-1') {
        this.cliente = await this.clientesService.getById(id);
        this.imagemSelecionada = this.cliente.uriFotoParaExibir;
    } else {
        this.cliente = {clienteid: '', nome: '', email: '', telefone: '',
renda: 0.00, nascimento: new Date(),
            uriFotoCapturada: '', nomeFotoEnviada: '', uriFotoParaExibir:
this.imagemSelecionada };
        this.modoDeEdicao = true;
    }

    this.clientesForm = this.formBuilder.group({
        clienteid: [this.cliente.clienteid],
        nome: [this.cliente.nome, Validators.required],
        email: [this.cliente.email, Validators.required],
        telefone: [this.cliente.telefone, Validators.required],
        renda: [this.cliente.renda, Validators.required],
        nascimento: [this.cliente.nascimento.toISOString(),
Validators.required],
        uriFotoCapturada: [this.cliente.uriFotoCapturada],
```

```
        nomeFotoEnviada: [this.cliente.nomeFotoEnviada],
        uriFotoParaExibir: [this.cliente.uriFotoParaExibir]
    });
}
```

Observou as alterações dos blocos `if` e `else`? No `if` corrigimos a propriedade para a atribuição a `this.imagemSelecionada` do cliente recuperado, se for uma consulta. Já no `else`, inicializamos as novas propriedades para o novo cliente que será inserido.

Outra alteração que certamente você notou está na definição do objeto `this.clientesForm`, que agora considera as três novas propriedades, dentro delas a substituição da propriedade `foto`, que não existe mais.

Seguindo a lógica do comportamento do usuário, vamos pensar agora na gravação dos dados, uma vez que a captura da imagem não sofrerá alterações. Veja o código a seguir, que deve ser implementado no método `submit()`, antes da invocação ao `update()`. Neste código, só alteramos a propriedade caso o cliente tenha capturado uma foto.

```
if (this.uriArquivoImagem) {

this.clientesForm.controls.uriFotoCapturada.setValue(this.uriArquivoImagem
);
}
```

Veja que estamos substituindo a instrução a seguir pelas anteriores. Desta maneira, retire-a de seu código.

```
this.clientesForm.controls.foto.setValue(this.imagemSelecionada);
```

Buscando finalizar esta alteração, vamos mudar então o método `update()` da classe `ClientesService`, tal qual é apresentado na sequência.

```
async update(cliente: Cliente): Promise<void> {
    try {
      if (cliente.uriFotoCapturada) {
```

```
        cliente.nomeFotoEnviada = this.criarNomeArquivoImagem();
        const blobDaImagem = await
this.criarBlobDeArquivoDeImagem(cliente.uriFotoCapturada);
        cliente.uriFotoParaExibir = await this.upload(blobDaImagem,
cliente.nomeFotoEnviada);
      }

      cliente.nascimento = new Date(cliente.nascimento);
      cliente.uriFotoCapturada = '';

      await
this.firestore.doc(`clientes/${cliente.clienteid}`).set(cliente);
    } catch (e) {
      console.error(e);
    }
}
```

Observou que levamos toda lógica de upload da foto para antes da atualização dos dados? É porque precisamos da URL do arquivo enviado para o `Storage` ligado ao nosso cliente, que também será enviado ao Firebase. O processo relacionado ao upload só ocorrerá se uma foto foi realmente selecionada pelo usuário. Atribuímos valores relacionados à imagem em propriedades do objeto `cliente` antes de enviar este objeto para o Firebase.

Se você digitou o código, pode se deparar com um erro relacionado à invocação do `upload()`, pois estamos esperando que ele retorne um dado, que é a URL, mas nossa implementação anterior não fazia isso. Vamos adaptar nosso método já existente. Veja-o na sequência, completo, para facilitar a leitura.

```
async upload(imagem, nomeImagem): Promise<string> {
    if (imagem) {
        try {
            nomeImagem = 'fotosClientes/' + nomeImagem;
            const resultadoUpLoad = await
firebase.storage().ref().child(nomeImagem).put(imagem);
            const urlUpLoad = await resultadoUpLoad.ref.getDownloadURL();
            return urlUpLoad;
          } catch (e) {
```

```
            console.log(e);
        }
    }
}
```

Verificou que nosso método retorna uma `Promise<string>` ? Veja também que a invocação ao `put()` tem agora o resultado atribuído a uma variável. Ela traz uma referência ao objeto enviado, que tem a URL de download recuperada pela invocação a `getDownloadURL()` atribuída à variável `urlUpLoad` , que por sua vez é retornada ao `update()` .

Com essas alterações já podemos realizar um novo teste em nossa aplicação. Entretanto, antes disso, recomendo que remova as imagens já enviadas ao `Storage` e os documentos enviados ao `FireStore` . Não é uma obrigatoriedade, mas mudamos nossa classe de modelo e isso não alterará a estrutura dos documentos já gravados, até que sejam novamente enviados.

Teste sua aplicação, crie um novo Cliente e capture uma foto para ele. Grave o cliente e veja que após a gravação, quando a aplicação retornar para a listagem de clientes, a imagem já aparece lá. Pode ser que você precise comentar momentaneamente seu código do método `removerCliente()` , que ainda faz referência à propriedade `foto` , removida de nosso modelo. Verifique a imagem enviada ao seu `Storage` e ao `FireStore` também.

8.8 A consulta, alteração e remoção da imagem

Você experimentou, no teste anterior, trocar uma imagem de um cliente que já tenha uma foto? Funciona certo? Mas você chegou a verificar no Storage que a imagem anterior não foi eliminada? Em nossa classe `ClientesService` , vamos criar o método a seguir para resolvermos este problema.

```
async deleteImagem(nomeImagem) {
    try {
      nomeImagem = 'fotosClientes/' + nomeImagem;
      await firebase.storage().ref().child(nomeImagem).delete();
    } catch (e) {
        console.log(e);
    }
}
```

Veja que uma vez mais estamos utilizando uma string constante para a composição do nome da imagem. Isso pode nos causar problemas no futuro, para o caso de precisarmos alterar. Uma dica que fica para você fazer em suas aplicações é ter este valor em uma constante pública e então utilizar esta constante e não o valor fixo.

A remoção da imagem é obtida pela execução da instrução chave, que invoca o método `delete()`. Que é bem semântico e semelhante ao envio de nosso arquivo. Mas onde invocar este método criado?

A resposta está no código a seguir, que é uma parte do `update()`. Observe um novo `if`, baseado no valor de `nomeFotoEnviada`. Caso haja um nome para a foto, quer dizer que uma já foi enviada para o Storage, o que nos leva a removê-la, invocando então nosso novo método. O resto do código é só para você identificar onde inserir essa nova implementação.

```
if (cliente.uriFotoCapturada) {
    if (cliente.nomeFotoEnviada) {
      this.deleteImagem(cliente.nomeFotoEnviada);
    }
    cliente.nomeFotoEnviada = this.criarNomeArquivoImagem();
    const blobDaImagem = await
this.criarBlobDeArquivoDeImagem(cliente.uriFotoCapturada);
    cliente.uriFotoParaExibir = await this.upload(blobDaImagem,
cliente.nomeFotoEnviada);
}
```

Altere a foto de um cliente já existente e acompanhe os arquivos no `Storage`. Veja que o antigo é removido e o novo inserido. Você já testou começar uma alteração, capturar uma nova imagem e depois

de ela ser exibida, cancelar a edição? Se fez, notou que a nova imagem continua sendo exibida. No método `cancelarEdicao()`, vamos inserir a instrução a seguir.

```
this.imagemSelecionada = this.cliente.uriFotoParaExibir;
```

Resta-nos a implementação que alterará a remoção de nossa imagem no Store quando o usuário remover o cliente. Já temos a remoção de cliente pronta, que é realizada no deslizar do cliente desejado na listagem dos clientes registrados. Inicialmente, vamos adaptar nosso método de remoção na classe `ClientesService`, tal qual é apresentado na sequência.

```
async removeById(clienteId: string, fotoEnviada: string): Promise<void> {
    await this.deleteImagem(fotoEnviada);
    return this.firestore.doc(`clientes/${clienteId}`).delete();
}
```

Recebemos agora o nome da foto que deverá ser removida e invocamos o método `deleteImagem()`, que implementamos anteriormente. Poderíamos pensar em variações para este método. Poderíamos recuperar o cliente por meio do ID recebido e então utilizar o nome da foto enviada, mas isso causaria mais tráfego. Poderíamos também pensar em tirar o `ById` do nome do método e receber todo o cliente e então utilizar as propriedades do objeto recebido. Tudo isso são sugestões para você.

Com a alteração anterior realizada, precisamos modificar também a invocação ao método `removeById()`, para enviar o novo argumento. Para isso, em nossa classe `ClientesListagemPage`, do arquivo `clientes-listagem.page.ts`, poderemos deixar nosso método `removerCliente` tal qual o código a seguir.

```
async removerCliente(cliente: Cliente) {
    try {
        const successFunction = async () => {
            this.clientesService.removeById(cliente.clienteid,
cliente.nomeFotoEnviada);
        };
```

```
        await this.alertService.presentConfirm('Remover Cliente',
'Confirma remoção?', successFunction);
    } catch (e) {
        await this.alertService.presentAlert('Falha', 'Remoção não foi
executada', e, ['Ok']);
    }
}
```

Já podemos testar nossa aplicação e ver se a remoção de um cliente também remove a foto dele no `Storage`. Fica a dica para você inserir um `Toasting` após o sucesso da remoção.

Conclusão

Trabalhamos aqui a captura de imagens, tanto do álbum de fotografias como utilizando a câmera do dispositivo. Gravamos a imagem capturada no próprio dispositivo, depois vimos a importância de termos as imagens na nuvem e fizemos isso por meio do `Storage` do `Firebase` do Google. Processo relativamente simples. Foi realmente um capítulo muito importante para o desenvolvimento de aplicações para dispositivos móveis conectadas à nuvem.

No próximo capítulo trabalharemos a autenticação de nosso usuário em nossa aplicação e consequentemente no Firebase, pois da maneira que temos a implementação, o acesso aos nossos recursos está público, não temos restrições.

Capítulo 9
Autenticação e criação de usuários no Firebase e uso de Tabs

Implementamos até o capítulo anterior o consumo de dois serviços do Firebase: o Firestore, para persistência remota de dados e o Storage, para armazenar fotos de nossos clientes. Até aí tudo bem, porém, não temos nenhum controle para limitar o acesso apenas a usuários de nossa aplicação.

Poderíamos pensar em desenvolver toda uma arquitetura para controle de usuários e validá-lo sempre para que estes dados possam ser acessados. Entretanto, o Firebase também nos oferece um serviço de autenticação muito bom e faremos uso dele neste capítulo.

Como um último recurso para navegação entre páginas para o usuário, veremos o `Tabs` , um controle que exibe na tela de seu dispositivo alguns botões, pelos quais o usuário terá acesso a páginas da aplicação. É outra maneira de navegação, além do Sidemenu que já conhecemos.

9.1 Preparação do projeto para a autenticação

Neste capítulo, vou optar por criarmos um projeto novo e nele implementar o que vamos utilizar. Para facilitar, veremos todos os passos aqui que serão necessários para nosso novo projeto.

Acesse o terminal, na pasta que armazena seus projetos e nela digite a instrução a seguir.

```
ionic start capitulo09 blank --type=angular --no-git
```

Como faremos uso do Firebase no projeto, precisamos instalar nele as dependências necessárias. Desta maneira, na pasta de seu projeto, digite a seguinte instrução e aguarde a instalação concluir.

```
npm install @angular/fire@5.2.1 firebase@6.1.1
```

Já temos nossa aplicação criada no Firebase, precisamos agora apenas configurar o processo de autenticação. Para isso, no menu de desenvolvimento de sua aplicação, lá no Console Firebase, clique na opção `Authentication` e, quando a nova página for exibida, no menu superior dela, clique em `Método de login` (Sign in method).

Com a nova página exibida, será possível identificar uma boa variedade de mecanismos de controle de autenticação a que o Firebase está integrado. Para nossa aplicação inicial deste capítulo, nós trabalharemos com `E-mail/senha` . Basta então clicarmos no ícone que aparece ao lado direito da opção quando colocamos o ponteiro do mouse sobre ela.

Quando formos ativar este método, uma janela será exibida, solicitando sua confirmação do que deseja ativar. Nós queremos apenas ativar o método de autenticação por email e senha. Habilite apenas esta opção. Veja a figura a seguir para lhe orientar, lembrando que ela pode mudar dependendo sempre da atualização da ferramenta do Google.

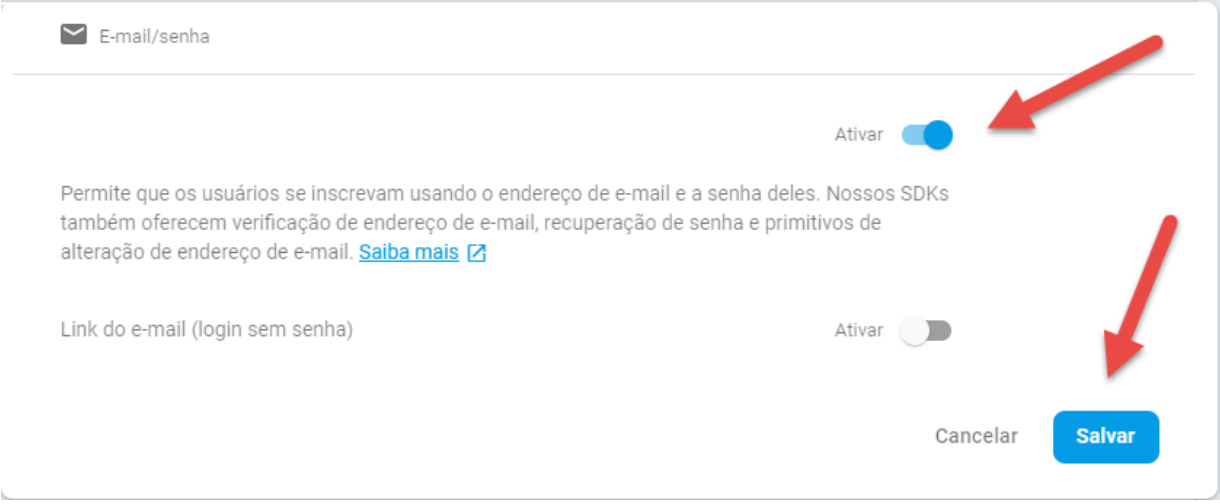

E-mail/senha
Ativar
Permite que os usuários se inscrevam usando o endereço de e-mail e a senha deles. Nossos SDKs também oferecem verificação de endereço de e-mail, recuperação de senha e primitivos de alteração de endereço de e-mail. Saiba mais
Link do e-mail (login sem senha)
Ativar
Cancelar
Salvar

Figura 9.1: Ativando autenticação por e-mail/senha

Precisaremos agora trazer para este nosso novo projeto as configurações de ambiente para conexão com o Firebase. As mesmas que utilizamos nos capítulos anteriores. Elas estão em um arquivo chamado `credentials.ts`, na pasta `/src/app`. Você pode copiar o arquivo para o mesmo caminho no novo projeto ou criar o arquivo e colar as configurações. Para relembrar, trago na sequência as configurações, que você deve adaptar para seus dados.

```
export const firebaseConfig = {
    production: false,
    firebase: {
        apiKey: 'BIzaSyCNi6RyIWWd2vNtfGDZSolWVPEI41fr3vc',
        authDomain: 'ionic-cc-fb.firebaseapp.com',
        databaseURL: 'https://ionic-cc-fb.firebaseio.com',
        projectId: 'ionic-cc-fb',
        storageBucket: 'ionic-cc-fb.appspot.com',
        messagingSenderId: '80326637376'
    }
};
```

9.2 O login para acessar a aplicação

Nosso primeiro passo para a implementação de nossa autenticação será a criação de nossa visão de login, pela qual o usuário informará um email e uma senha, que serão validadas e então o acesso poderá ser liberado, ou não. No terminal, digite a seguinte instrução.

```
ionic generate page login
```

Com a página criada, podemos eliminar o arquivo `login.page.spec.ts`, pois é para testes e não estamos vendo isso. Podemos também aproveitar e eliminar a página `home`, pois nossa página principal, de início da aplicação será a `login`. Para finalizar o processo e termos nossa página configurada como de início, vamos alterar nossas rotas no `app-routing.module.ts`, tal qual o código a seguir. Retiramos os inicialmente criados com a aplicação.

```
const routes: Routes = [
  { path: '', loadChildren: './login/login.module#LoginPageModule' },
];
```

Vamos trabalhar agora nossa classe que renderizará o template para o usuário, a `login.page.ts`, que tem a primeira parte do código apresentada na sequência, que você deve implementar em seu projeto.

```
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormBuilder, Validators, FormControl } from
'@angular/forms';
import { NavController } from '@ionic/angular';

@Component({
  selector: 'app-login',
  templateUrl: './login.page.html',
  styleUrls: ['./login.page.scss'],
})
export class LoginPage implements OnInit {

  validationsForm: FormGroup;
  errorMessage: string;

  validationMessages = {
```

```
    email: [
      { type: 'required', message: 'Informe o E-Mail' },
      { type: 'pattern', message: 'Informe um email válido' }
    ],
    password: [
      { type: 'required', message: 'Informe a senha' },
      { type: 'minlength', message: 'A senha precisa ter ao menos 6
caracteres' }
    ]
  };
}
```

Vamos à segunda parte do código? Veja-a na sequência. Inseri o cabeçalho da classe apenas para que a indentação ficasse correta.

```
export class LoginPage implements OnInit {

  // Código apresentado anteriormente

  constructor(
    private formBuilder: FormBuilder
  ) { }

  ngOnInit() {
    this.validationsForm = this.formBuilder.group({
      email: new FormControl('', Validators.compose([
        Validators.required,
        Validators.pattern('^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+.[a-zA-Z0-
9-.]+$')
      ])),
      password: new FormControl('', Validators.compose([
        Validators.minLength(6),
        Validators.required
      ])),
    });
  }
}
```

Temos a injeção de um `NavController` no construtor, que já estamos utilizando no `ngOnInit()` para a criação dos controles no `Form`. Verifique as regras que devem ser aplicadas nos controles `email` e

`password` . Com esta implementação realizada, podemos seguir para nosso template, o `login.page.html` . Veja atenciosamente o código na sequência.

```
<ion-header>
  <ion-toolbar color="primary">
    <ion-title>Login</ion-title>
  </ion-toolbar>
</ion-header>

<ion-content padding>
  <form class="form" [formGroup]="validationsForm">
    <ion-item>
        <ion-label  position="floating" color="primary">Email</ion-label>
        <ion-input type="text" formControlName="email"></ion-input>
    </ion-item>

    <div>
      <ng-container *ngFor="let validation of validationMessages.email">
        <div class="validation-errors"
*ngIf="validationsForm.get('email').hasError(validation.type) &&
(validationsForm.get('email').dirty ||
validationsForm.get('email').touched)">
          {{ validation.message }}
        </div>
      </ng-container>
    </div>

    <ion-item>
      <ion-label  position="floating" color="primary">Password</ion-label>
      <ion-input type="password" formControlName="password" class="form-
controll" required></ion-input>
    </ion-item>
    <div class="validation-errors">
      <ng-container *ngFor="let validation of
validationMessages.password">
        <div class="error-message"
*ngIf="validationsForm.get('password').hasError(validation.type) &&
(validationsForm.get('password').dirty ||
validationsForm.get('password').touched)">
```

```
          {{ validation.message }}
        </div>
      </ng-container>
    </div>

    <div style="text-align: center" padding>
      <ion-button  class="submit-btn" type="submit"
[disabled]="!validationsForm.valid">Acessar</ion-button>
    </div>

    <div style="text-align: center">
      <label class="validation-errors">{{errorMessage}}</label>
    </div>
  </form>
  <p style="text-align: center">
    Não tem conta ainda? <a>Crie uma conta</a>
  </p>
</ion-content>
```

Precisamos criar nosso arquivo de estilos, com o nome `login.page.scss` e nele vamos inserir o código a seguir.

```
.validation-errors {
    text-align: center;
    color: red;;
}
```

Para que possamos testar nossa aplicação e vermos nosso formulário de login renderizado, precisamos importar `ReactiveFormsModule` em nosso arquivo `app.module.ts` e `login.module.ts` . Veja o código a seguir, com a orientação de onde inserir.

```
@NgModule({
  imports: [
    // Ocultado
    ReactiveFormsModule
  ],
  // Ocultado
})
```

A figura a seguir mostra o login em nosso pseudoemulador. Você pode digitar valores nos campos para validar as mensagens de erro.

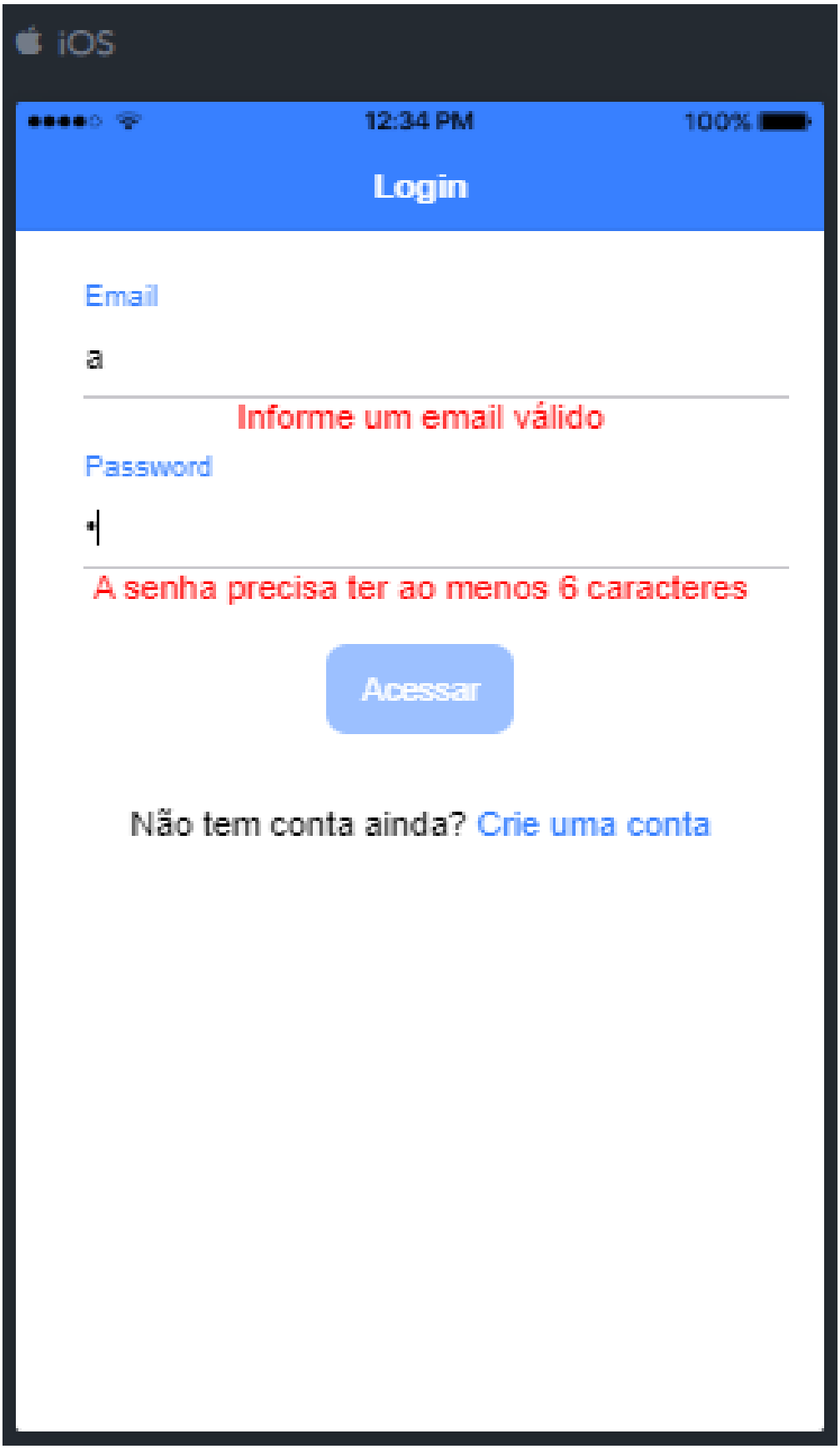

iOS
12:34 PM
100%
Login
Email
a
Informe um email válido
Password
A senha precisa ter ao menos 6 caracteres
Acessar
Não tem conta ainda? Crie uma conta

Figura 9.2: Página de login

Vamos agora implementar a funcionalidade que validará os dados de email e senha do usuário para autenticação no Firebase. Vamos trazer para cá novamente o uso de serviços, tal qual vínhamos fazendo para a persistência de dados. Acesse o terminal e digite a instrução a seguir.

```
ionic g service services/autenticacao
```

Vamos agora a algumas implementações para este nosso serviço. A primeira delas é importarmos recursos do Firebase para a classe. Vamos então inserir a instrução a seguir no início do arquivo.

```
import * as firebase from 'firebase/app';
```

Na sequência, precisamos importar nosso método responsável pela tentativa de autenticação do usuário no Firebase. Veja-o na sequência, com algumas explicações logo após.

```
login(value) {
    return new Promise<any>((resolve, reject) => {
      firebase.auth().signInWithEmailAndPassword(value.email,
value.password)
      .then(
        res => resolve(res),
        err => reject(err));
    });
}
```

Percebeu que não temos nada de complexo no método? Receberemos um objeto, que terá `email` e `password` como propriedades. Enviamos estes valores para um método do Firebase chamado `signInWithEmailAndPassword()`, retornado por `auth()`. Temos retornos relativos ao sucesso, pelo `resolve()` e em caso de erro o `reject()`.

Seguindo o fluxo para a implementação, precisamos invocar este método em nossa classe que renderizará o template, a `LoginPage`, que está no arquivo `login.page.ts`. Precisamos injetar nosso serviço

em nossa classe, inserindo a instrução a seguir em nosso construtor.

```
private autenticacaoService: AutenticacaoService
```

Precisaremos implementar também um método em nossa classe da página, que será invocado pelo template. Veja este método na sequência.

```
login(value) {
    this.autenticacaoService.login(value)
    .then(res => {
      this.errorMessage = 'Autenticado';
    }, err => {
      this.errorMessage = err.message;
    });
}
```

No código anterior, invocamos o método `login` de nosso serviço, enviando a ele um valor que será recebido por este método. Em caso de sucesso, será escrito `Autenticado` no controle que exibe erros de conexão, apenas para teste. Depois trabalharemos o redirecionamento para uma página da aplicação. Em caso de erro, a mensagem recuperada pelo erro será renderizada. Será uma mensagem padrão do Firebase. Você pode trabalhar essas mensagens e exibi-las de acordo com sua necessidade.

Precisamos possibilitar agora que nosso usuário requisite este método quando interagir com o botão de acesso à aplicação. Vamos adaptar nosso elemento `<form>` do template, tal qual é apresentado na sequência, com o *binding* de evento para `ngSubmit` .

```
<form class="form" [formGroup]="validationsForm"
(ngSubmit)="login(validationsForm.value)">
```

Para que possamos testar nossa aplicação, precisamos iniciar o serviço do Firebase em nossa aplicação, que ainda não fizemos. Desta maneira, em nosso arquivo `app.module.ts` , antes de `@NgModule` , insira as instruções a seguir.

```
import { firebaseConfig } from './credentials';
import * as firebase from 'firebase';

firebase.initializeApp(firebaseConfig.firebase);
```

Vamos testar novamente nossa aplicação? Tente informar um email e senha e então clique no botão acessar. Notou o erro? No link https://firebase.google.com/docs/reference/js/firebase.auth.Auth.html#signinwithemailandpassword você pode verificar, quando quiser, algumas orientações sobre como identificar os erros possíveis na invocação do método `signInWithEmailAndPassword()`.

Com o que temos até agora, é preciso que você crie um usuário em seu Firebase Console. Basta acessar o menu `Authentication` depois na opção `Usuários` e então em `Adicionar Usuário`. Depois disso, teste com as credenciais válidas e veja a mensagem de sucesso abaixo do botão.

9.3 A aplicação acessível por Tabs

Quando nosso usuário é autenticado, o correto é que ele seja direcionado a uma página da aplicação. Pode ser uma página apenas, pode ser o Sidebar Menu que implementamos anteriormente, ou ainda, uma página que possua outras páginas, acessíveis via `Tabs`, que é o foco desta seção.

Começaremos com a criação da página inicial de nossa aplicação, que será exibida após a autenticação correta por parte do usuário e a chamaremos de `home`. Mas, como criamos nosso projeto do início, por padrão o Ionic já cria a página `home`. Vamos substituir o código que temos no template pelo que tem a seguir.

```
<ion-tabs>
  <ion-tab-bar slot="bottom">
    <ion-tab-button tab="pecas">
      <ion-icon name="pricetags"></ion-icon>
```

```
      <ion-label>Peças</ion-label>
    </ion-tab-button>

    <ion-tab-button tab="clientes">
      <ion-icon name="person"></ion-icon>
      <ion-label>Clientes</ion-label>
    </ion-tab-button>

    <ion-tab-button tab="atendimentos">
      <ion-icon name="car"></ion-icon>
      <ion-label>Atendimentos</ion-label>
    </ion-tab-button>

    <ion-tab-button (click)="logout()">
      <ion-icon name="log-out"></ion-icon>
      <ion-label>Sair</ion-label>
    </ion-tab-button>
  </ion-tab-bar>
</ion-tabs>
```

A página `home` será a página que abrigará o menu com `Tabs` acessíveis para o usuário. Essa definição começa com `<ion-tabs>`, que contém a barra de opções que será exibida, configurada pelo componente `<ion-tab-bar>`, que é composto de `<ion-tab-button>`, representando cada opção acessível.

Em `<ion-tab-button>` temos `tab`, que define um nome que será utilizado como rota para o Angular poder redirecionar a exibição da página correta dentro da página `home`. O último `<ion-tab-button>` não tem um `tab`, mas está ligado ao evento `click`, que invocará um método chamado `logout()`.

Com estas informações, podemos implementar agora nossa classe ligada ao template `home`. Veja o código dela na sequência.

```
import { Component, OnInit, ViewChild } from '@angular/core';
import { AutenticacaoService } from '../services/autenticacao.service';
import { NavController, IonTabs } from '@ionic/angular';

@Component({
```

```
  selector: 'app-home',
  templateUrl: './home.page.html',
  styleUrls: ['./home.page.scss'],
})
export class HomePage implements OnInit {

  constructor(
    private autenticacaoService: AutenticacaoService,
    private navCtrl: NavController
  ) { }

  ngOnInit() { }

  logout() {
    this.autenticacaoService.logout()
    .then(res => {
      console.log(res);
      this.navCtrl.navigateBack('');
    })
    .catch(error => {
      console.log(error);
    });
  }
}
```

No método `logout()` da listagem anterior, apenas invocamos o método `logout()` que precisamos implementar em nosso `AutenticacaoService`, que você pode verificar na listagem a seguir.

```
logout() {
 return new Promise((resolve, reject) => {
   if (firebase.auth().currentUser) {
     firebase.auth().signOut()
     .then(() => {
       resolve();
     }).catch((error) => {
       reject();
     });
   }
 });
}
```

Para o `logout()`, seguimos a mesma lógica adotada para o `login()`, que se resume a invocar um método de `auth()` no Firebase. Entretanto, note que este método só é invocado se tivermos um usuário autenticado, verificado por `currentUser`. Como em `login()`, o método retorna `resolve()` ou `reject()`.

Mas como chegaremos em `home`? Ao criarmos nossa página, a rota a seguir já foi configurada em nosso `app-routing.module.ts`, basta invocá-la no sucesso da autenticação.

```
{ path: 'home', loadChildren: './home/home.module#HomePageModule' }
```

Vamos agora rever nosso código de `login()` da página de `login`. Veja sua nova implementação na sequência. Temos só duas alterações, mas preferi trazê-lo todo.

```
login(value) {
    this.autenticacaoService.login(value)
    .then(res => {
      this.errorMessage = '';
      this.navCtrl.navigateForward('/home');
    }, err => {
      this.errorMessage = err.message;
    });
}
```

Notou que tiramos o texto da atribuição a `this.errorMessage` e que inserimos a instrução `this.navCtrl.navigateForward('/home');`? Mas de onde vem este `navCtrl`? Precisamos injetá-lo em nosso construtor com o código a seguir.

```
private navCtrl: NavController
```

Para testarmos nossa aplicação, vamos criar uma página básica para Peças com a instrução a seguir no terminal. No template, você pode colocar um texto qualquer dentro de `<ion-content>`, apenas para ver a execução da aplicação.

```
ionic generate page pecas
```

Mas isso ainda não é o suficiente. Lembra que falei sobre o elemento `tab` dentro do `<ion-tab-button>` ? Pois bem, a aplicação buscará por uma rota chamada `pecas` e não a temos. Se você lembrar da implementação do Sidemenu, tivemos que implementar um sistema de rota específico para ele. Precisaremos fazer o mesmo aqui, pois a navegação é outra após a autenticação. Não pode mais levar em consideração as rotas do `app-routing.module.ts` . Precisamos então criar, na pasta `home` , o arquivo `home-routing.module.ts` , tal qual o código a seguir.

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomePage } from './home.page';
import { PecasPage } from '../pecas/pecas.page';

const routes: Routes = [
    {
      path: 'tabsMenu',
      component: HomePage,
      children: [
        {
          path: 'pecas',
          children: [
            {
              path: '',
              loadChildren: '../pecas/pecas.module#PecasPageModule'
            }
          ]
        },
      ]
    },
    {
      path: '',
      redirectTo: 'tabsMenu/pecas',
      pathMatch: 'full'
    }
  ];

@NgModule({
  imports:
```

```
    [
      RouterModule.forChild(routes)
    ],
  exports:
    [
      RouterModule
    ]
})
export class HomePageRoutingModule {}
```

Notou que o código é semelhante ao que fizemos para o Sidemenu? Antes de testarmos, vamos ao nosso `home.module.ts` e importar no `@NgModule()` nosso `PecasPageModule` .

Precisamos agora adaptar nosso `home.module.ts` para oferecer este módulo de roteamento que criamos. Sendo assim, deixe o `@NgModule()` dele tal qual apresento na sequência.

```
@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    IonicModule,
    HomePageRoutingModule
  ],
  declarations: [HomePage]
})
```

Veja a figura a seguir que apresenta a página com com os `Tabs` e e página de Peças renderizada.

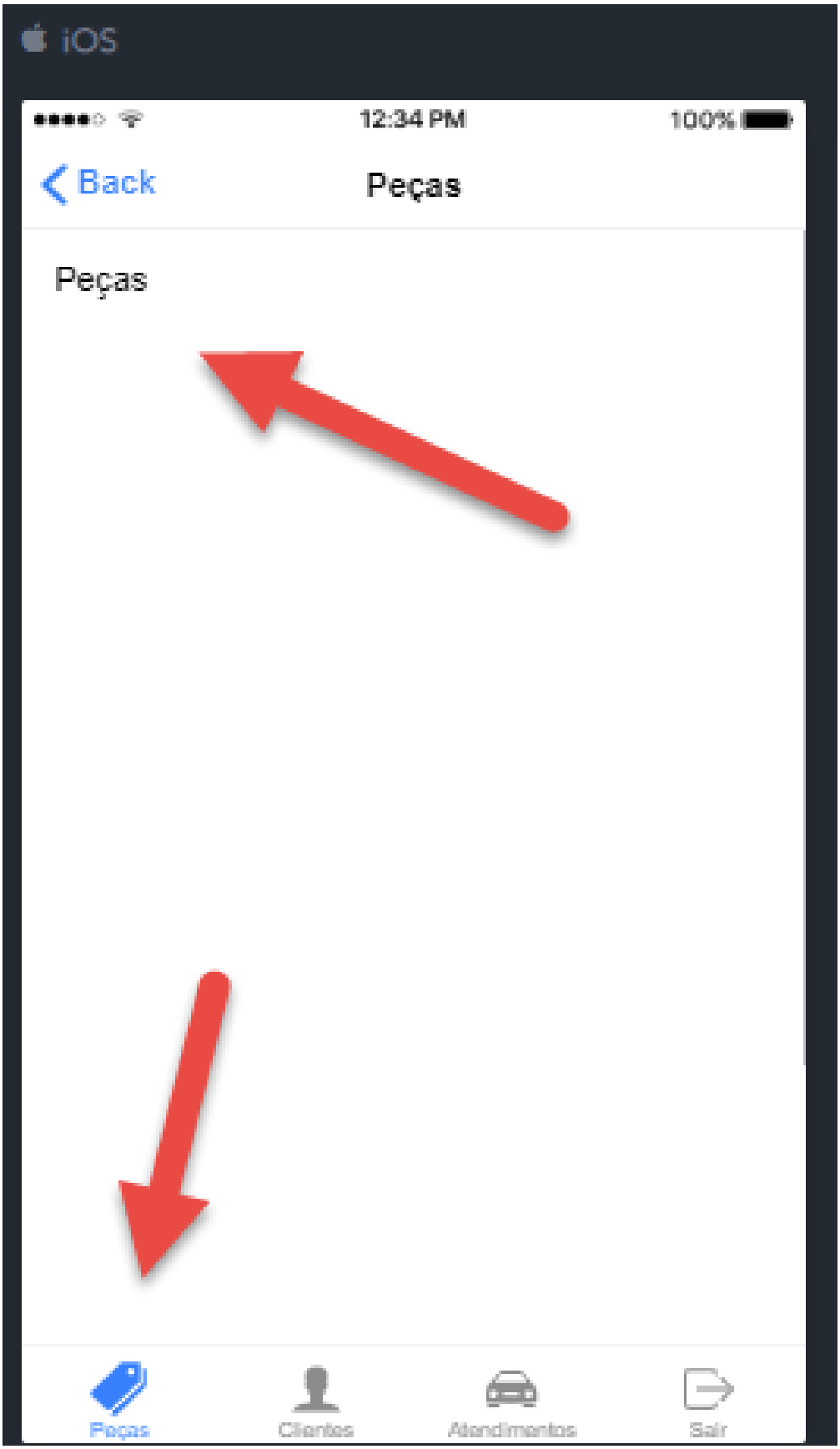

iOS
12:34 PM
100%
Back
Peças
Peças
Peças
Clientes
Atendimentos
Sair

Figura 9.3: Página com Tabs e a de Peças renderizada

9.4 A página para listagem de clientes registrados

Para testarmos o acesso ao conteúdo do Firebase apenas quando nosso usuário estiver conectado, vamos trazer para este novo projeto nosso componente de listagem de clientes, que implementamos no capítulo anterior. Pode copiar a pasta `listagem` dentro de `clientes` para uma pasta chamada `clientes` dentro do nosso projeto deste capítulo. Para auxiliar, trago na sequência as implementações destes arquivos. Foram mantidas apenas as instruções relativas à listagem.

clientes-listagem.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { IonicModule } from '@ionic/angular';
import { ClientesListagemPage } from './clientes-listagem.page';
import { Routes, RouterModule } from '@angular/router';

const routes: Routes = [
  {
    path: '',
    component: ClientesListagemPage
  }
];

@NgModule({
    imports: [
      CommonModule,
      FormsModule,
      IonicModule,
      RouterModule.forChild(routes)
    ],
```

```
    declarations: [
        ClientesListagemPage
    ]
})

export class ClientesListagemPageModule {}
```

clientes-listagem.page.html

```
<ion-header>
    <ion-toolbar>
      <ion-buttons slot="start">
          <ion-menu-button></ion-menu-button>
      </ion-buttons>
      <ion-title>Clientes</ion-title>
    </ion-toolbar>
</ion-header>

<ion-content padding-left padding-top>
  <ion-list>
    <ion-item-sliding *ngFor="let cliente of clientes | async">
      <ion-item>
        <ion-thumbnail slot="start">
            <ion-img [src]="cliente.uriFotoParaExibir"></ion-img>
        </ion-thumbnail>
        <ion-grid>
          <ion-row>
            <ion-label><p style="font-size: larger; font-weight: bold;">
{{cliente.nome}}</p></ion-label>
          </ion-row>
          <ion-row>
            <ion-col col-12>
              <ion-label><p style="font-weight: 400; text-align:
right">Telefone: {{cliente.telefone}}</p></ion-label>
            </ion-col>
          </ion-row>
        </ion-grid>
      </ion-item>

      <ion-item-options side="end">
        <ion-item-option color="danger">
```

```
          <ion-button color="danger" no-padding>
              <ion-icon name="trash"></ion-icon>
              Remover
          </ion-button>
        </ion-item-option>
      </ion-item-options>
    </ion-item-sliding>
  </ion-list>

  <ion-fab vertical="top" horizontal="end" slot="fixed" edge="true">
    <ion-fab-button>
      <ion-icon name="add"></ion-icon>
    </ion-fab-button>
  </ion-fab>
</ion-content>
```

clientes-listagem.page.ts

```
import { Component, OnInit } from '@angular/core';
import { Cliente } from 'src/app/models/cliente.model';
import { ClientesService } from 'src/app/services/clientes.service';
import { Observable } from 'rxjs';

@Component({
    templateUrl: './clientes-listagem.page.html'
})
export class ClientesListagemPage implements OnInit {

    public clientes: Observable<Cliente[]>;

    constructor(
        private clientesService: ClientesService,
        ) { }

    ngOnInit() {
    }

    ionViewWillEnter() {
        this.clientes = this.clientesService.getAll().valueChanges();
    }
}
```

Sabemos que junto desta página temos outros componentes ligados e precisamos também trazê-los para nossa aplicação. São eles:

- `clientes.service.ts` , que está na pasta `services` e deverá ser trazido para a pasta de mesmo nome;
- `cliente.model.ts` , que está na pasta `models` e deverá ser trazido para uma pasta de mesmo nome, a ser criada na aplicação deste capítulo.

Para facilitar, trago na sequência estes códigos também, apenas com o que será pertinente para este momento.

clientes.service.ts

```
import { Injectable } from '@angular/core';
import { Cliente } from '../models/cliente.model';
import { AngularFirestore, AngularFirestoreCollection } from
'@angular/fire/firestore';

@Injectable({
    providedIn: 'root'
})
export class ClientesService {
  constructor(
    private firestore: AngularFirestore
  ) { }

  public getAll(): AngularFirestoreCollection<Cliente> {
    return this.firestore.collection('clientes', ref =>
ref.orderBy('nome', 'asc'));
  }
}
```

cliente.model.ts

```
export interface Cliente {
    clienteid: string;
    nome: string;
    email: string;
    telefone: string;
    renda: number;
```

```
    nascimento: Date;
    uriFotoCapturada: string;
    nomeFotoEnviada: string;
    uriFotoParaExibir: string;
}
```

Com a codificação para a página de listagem de clientes concluída, precisamos agora registrar a rota para ele em nosso arquivo `home-routing.module.ts` , logo após a rota para `peças` . Veja isso na sequência.

```
,
{
    path: 'clientes',
    children: [
      {
        path: '',
        loadChildren: '../clientes/listagem/clientes-
listagem.module#ClientesListagemPageModule'
      }
    ]
},
```

Como agora nossa aplicação faz uso do Firestore, precisamos importar em nosso `app.module.ts` os componentes necessários. Sendo assim, insira as instruções a seguir ao final de `imports` do `@NgModule()` .

```
AngularFireModule.initializeApp(firebaseConfig),
AngularFirestoreModule
```

Caso no capítulo anterior tenha sido gravado algum cliente sem foto, precisamos trazer para nosso projeto a figura que utilizamos como padrão para o caso de ausência de uma foto. Sendo assim, traga a figura `icon_clientes.png` da pasta `assets/imgs` para nossa aplicação deste capítulo, para o mesmo endereço e com o mesmo nome.

Muito bem, agora já podemos testar nossa aplicação. Veja na figura a seguir a representação obtida para esta implementação.

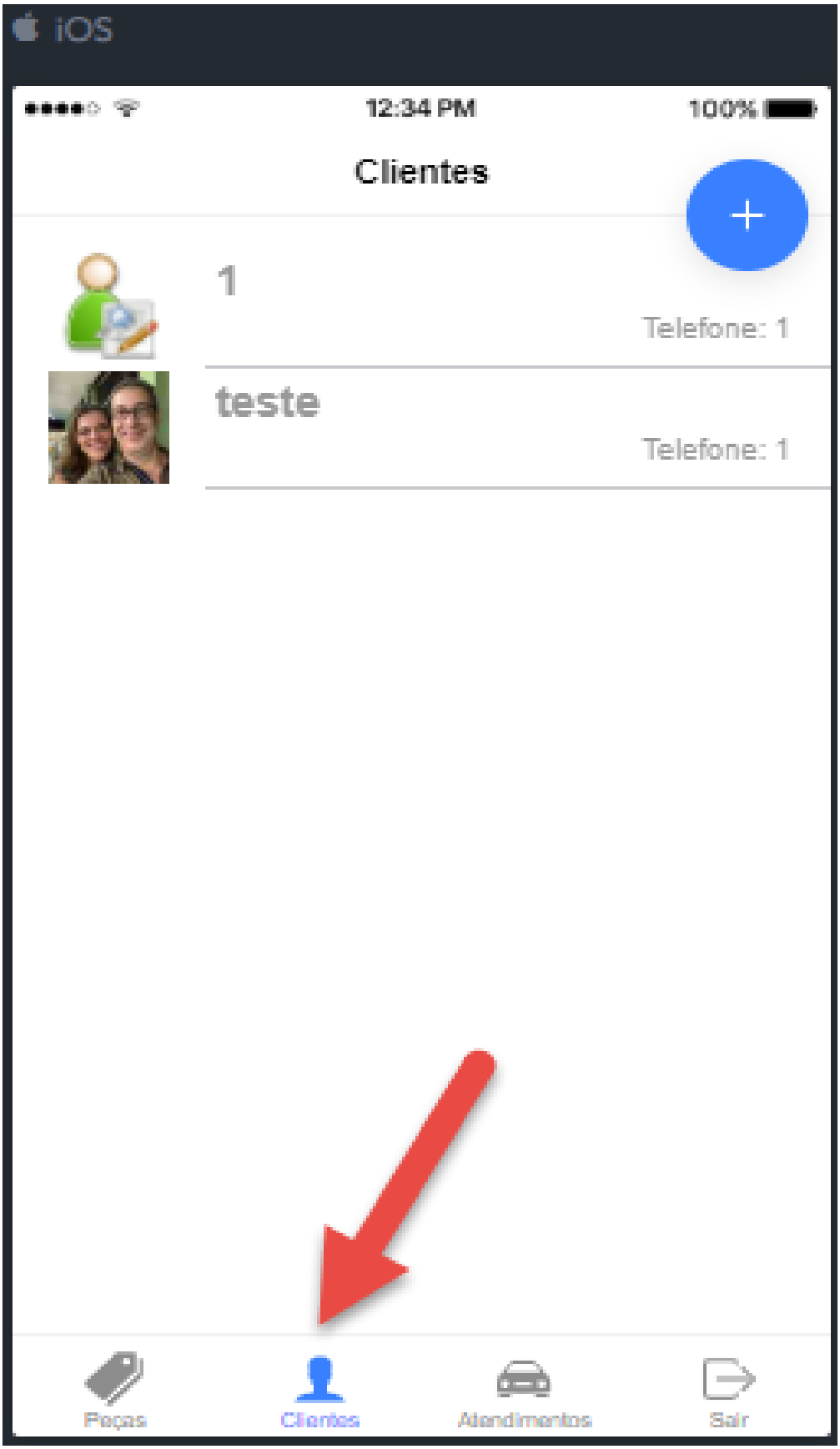

iOS
12:34 PM
100%
Clientes
1
Telefone: 1
teste
Telefone: 1
Peças
Clientes
Atendimentos
Sair

Figura 9.4: Página com Tabs e a de Clientes renderizada

9.5 Proibindo o acesso anônimo

Com a configuração que temos para acesso em nossa aplicação no Firebase, quer seja o Firestore ou o Storage, qualquer usuário com as credenciais acessa e não é isso que queremos. Precisamos corrigir isso, atribuindo uma regra mínima de autenticação para o acesso. Vamos então acessar o menu `Database` e depois `Regras` e vamos substituir a regra que lá está pela informada na sequência. A atualização e disponibilização para sua aplicação pode demorar alguns momentos.

```
allow read, write: if request.auth.uid != null;
```

Vamos realizar o mesmo procedimento para o `Storage`, inserindo em regras o mesmo código apresentado anteriormente. Para testarmos, vamos alterar nossa rota padrão, que está no `app-routing.module.ts` para não forçar a tela de login de início, indo direto para nossa página dos `Tabs`. Veja os códigos na sequência.

```
const routes: Routes = [
  // { path: '', loadChildren: './login/login.module#LoginPageModule' },
  // { path: 'home', loadChildren: './home/home.module#HomePageModule' }
  { path: '', loadChildren: './home/home.module#HomePageModule' }
];
```

Agora teste sua aplicação e escolha a tab `Clientes`. Ative a janela de desenvolvedor no navegador (pressionando F12) e veja na console o erro apresentado na figura a seguir.

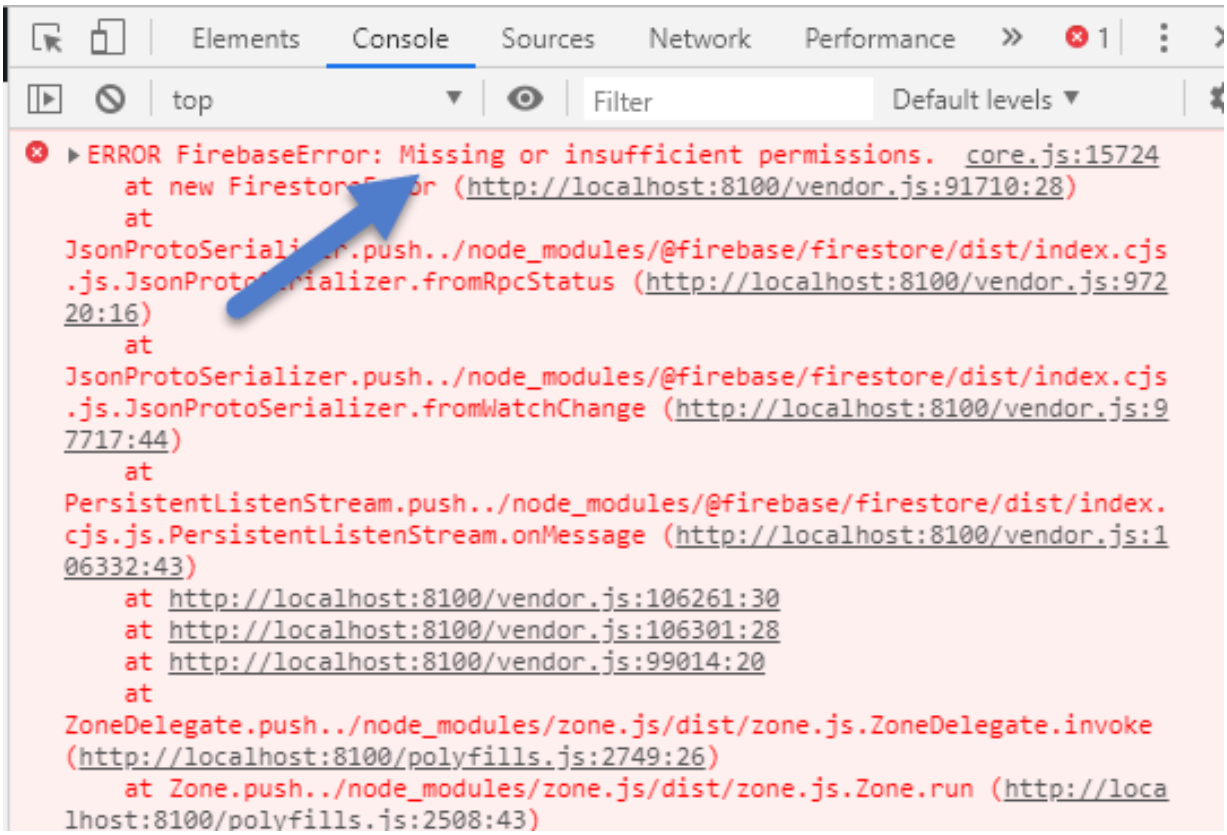


Figura 9.5: Erro ao acessar o Firebase sem autenticação

Nossos dados estão protegidos agora, podendo ser acessados apenas para quem se autenticar. Precisamos então adaptar nossa aplicação para que leve isso em consideração. Vamos começar estas alterações na classe `AutenticacaoService`, inserindo nela o método a seguir, que nos retornará um valor booleano informando se temos ou não um usuário autenticado.

```
public isUserAuthenticated(): boolean {
  return( (firebase.auth().currentUser !== null) );
}
```

Na sequência, precisamos alterar nosso método `getAll()` the `ClientesService`, para que, antes de realizar a leitura dos dados, verifique se nosso usuário está autenticado. Como consumiremos o novo método que implementamos pela listagem anterior, vamos realizar a injeção de `AutenticacaoService` em nosso construtor de `ClientesService`, com a instrução a seguir.

```
private autenticacaoService: AutenticacaoService
```

Agora sim, vamos consumir o novo método. Veja a nova implementação para `getAll()` na sequência. Observe que preferimos trabalhar com exceções caso não haja autenticação no Firebase.

```
public getAll(): AngularFirestoreCollection<Cliente> {
    if (!this.autenticacaoService.isUserAuthenticated()) {
      throw new Error('Usuário não está autenticado');
    }
    return this.firestore.collection('clientes', ref =>
ref.orderBy('nome', 'asc'));
}
```

E como fica nossa classe que renderiza o template visual? É nela que temos a invocação do `getAll()` que retornará os clientes a serem exibidos. Vamos trabalhar na situação em que, caso uma exceção seja disparada, um alerta será exibido ao usuário. Caso não haja exceção, os dados serão atribuídos e então exibidos.

Como comentei em alerta, vamos utilizar nossa classe `AlertService`, que já temos implementada de capítulos anteriores. Você pode trazer o arquivo para a pasta `services` de nossa aplicação. Para facilitar, trago o código necessário, que deve ser implementado no arquivo `alert.service.ts`, logo na sequência.

```
import { Injectable } from '@angular/core';
import { AlertController } from '@ionic/angular';

@Injectable({
    providedIn: 'root'
  })
export class AlertService {
    constructor(private alertCtrl: AlertController) {}

    async presentAlert(header: string, subHeader: string, message: string,
buttons: string[]) {
        const alert = await this.alertCtrl.create({
          header,
```

```
            subHeader,
            message,
            buttons
        });
        await alert.present();
    }
}
```

Por fim, vamos adaptar nosso método `ionViewWillEnter()` da listagem de clientes para esta nova situação. Veja-o na listagem a seguir.

```
ionViewWillEnter() {
    try {
        this.clientes = this.clientesService.getAll().valueChanges();
    } catch (error) {
        this.alertService.presentAlert('Erro de conexão', 'Usuário não
está autenticado', 'Tente conectar novamente', [ 'OK'] );
    }
}
```

Lembre-se de que precisamos injetar `alertService` em nosso construtor, com a instrução a seguir.

```
private alertService: AlertService
```

Agora podemos testar nossa aplicação. Veja que o alerta é exibido, pois não temos autenticação de usuário. Retorne as rotas do `app-routing.module.ts` ao normal e teste o acesso autenticado.

9.6 Sair da aplicação por meio de um logout

Temos ainda duas funcionalidades para implementar em relação à autenticação proposta para este capítulo. A de criação de um novo usuário e a de logout. Essa segunda é bem simples e a implementaremos em poucas linhas. Vamos começar com o método a seguir, na classe `AutenticacaoService` .

```
logout() {
 return new Promise((resolve, reject) => {
   if (firebase.auth().currentUser) {
     firebase.auth().signOut()
     .then(() => {
       resolve();
     }).catch((error) => {
       reject();
     });
   }
 });
}
```

Notou que uma vez mais nos baseamos na existência de um usuário autenticado? Caso exista, invocamos `signOut()` de `auth()`. Tudo muito simples e funcional.

Agora, precisamos consumir este método. Lembra que, em nosso menu de `Tabs`, o último botão não tem a propriedade `tab`? Vamos ligar um método com o evento `click` deste botão, tal qual mostro a seguir.

```
<ion-tab-button (click)="logout()">
```

Na sequência, na classe `HomePage`, precisamos implementar nosso método ligado anteriormente e que consumirá o serviço que também implementamos a pouco. Veja-o na sequência. Bem simples de entender, concorda?

```
logout() {
    this.autenticacaoService.logout()
    .then(res => {
      this.navCtrl.navigateBack('');
    })
    .catch(error => {
      console.log(error);
    });
}
```

Veja que após promovermos o redirecionamento para a página de login, por meio do `navCtrl`, precisamos injetar em nosso construtor, com a instrução a seguir.

```
private navCtrl: NavController
```

Vamos testar? Autentique-se, navegue até clientes e pressione o botão `Sair`. Lembre-se de que precisa ajustar a rota no `app-routing.module.ts`, que mudamos para nosso teste.

9.7 Criação de novos usuários diretamente pela aplicação

É muito comum ao acessarmos sites ou aplicativos em nossos dispositivos e eles nos solicitarem autenticação e, em conjunto, a possibilidade de criarmos uma nova conta para este acesso. O Firebase Authentication nos oferece de maneira bem simples este recurso e é o que implementaremos nessa seção. Vamos começar pelo método em nossa classe `AutenticacaoService`, que está na sequência.

```
registrar(value) {
 return new Promise<any>((resolve, reject) => {
   firebase.auth().createUserWithEmailAndPassword(value.email,
value.password)
   .then(
     res => resolve(res),
     err => reject(err));
 });
}
```

Notou o quão facilitadora é a API oferecida pelo Firebase? Apenas invocamos um método, enviando a ele o email e senha para a conta que precisará ser criada.

Agora precisamos criar nossos arquivos necessários para esta nova página, que se chamará `registrar`. Para terminar, vamos implementar a instrução a seguir? Com a aplicação em execução, elimine, se quiser, o arquivo de teste, como estamos fazendo desde o início. A instrução criará nossos arquivos dentro de uma pasta chamada `usuarios`.

```
ionic generate page usuarios/registrar
```

Trabalharemos alguns estilos em nosso template, vamos então já codificá-los em nosso arquivo `registrar.page.scss`:

```
.validation-errors {
    text-align: center;
    color: red;;
}

.validation-success {
    text-align: center;
    color: green;;
}
```

Podemos agora implementar nosso template.

```
<ion-header>
  <ion-toolbar color="primary">
    <ion-title>Registrar usuário</ion-title>
  </ion-toolbar>
</ion-header>

<ion-content padding>
  <form class="form" [formGroup]="validationsForm"
(ngSubmit)="registrar(validationsForm.value)">

    <ion-item>
      <ion-label  position="floating" color="primary">Email</ion-label>
      <ion-input type="text" formControlName="email"></ion-input>
    </ion-item>
    <div class="validation-errors">
      <ng-container *ngFor="let validation of validationMessages.email">
        <div class="error-message"
```

```
*ngIf="validationsForm.get('email').hasError(validation.type) &&
(validationsForm.get('email').dirty ||
validationsForm.get('email').touched)">
          {{ validation.message }}
        </div>
      </ng-container>
    </div>

    <ion-item>
      <ion-label  position="floating" color="primary">Senha</ion-label>
      <ion-input type="password" formControlName="password" class="form-
controll" required></ion-input>
    </ion-item>
    <div class="validation-errors">
      <ng-container *ngFor="let validation of
validationMessages.password">
        <div class="error-message"
*ngIf="validationsForm.get('password').hasError(validation.type) &&
(validationsForm.get('password').dirty ||
validationsForm.get('password').touched)">
          {{ validation.message }}
        </div>
      </ng-container>
    </div>

    <div style="text-align: center">
      <ion-button  class="submit-btn" type="submit"
[disabled]="!validationsForm.valid">Registrar</ion-button>
      <br/><label class="validation-errors">{{errorMessage}}</label>
      <br/><label class="validation-success">{{successMessage}}</label>
    </div>
  </form>
  <div style="text-align: center">
    <p class="go-to-login">Já tem uma conta? <a
(click)="goLoginPage()">Tente se autenticar</a></p>
  </div>
</ion-content>
```

Para renderizarmos nosso template, precisamos implementar nossa classe em TypeScript e podemos ver o código dela na sequência.

```
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormBuilder, Validators, FormControl } from
'@angular/forms';
import { AutenticacaoService } from '../../services/autenticacao.service';
import { NavController } from '@ionic/angular';

@Component({
  selector: 'app-register',
  templateUrl: './registrar.page.html',
  styleUrls: ['./registrar.page.scss'],
})
export class RegistrarPage implements OnInit {

  validationsForm: FormGroup;
  errorMessage: '';
  successMessage: string;

  validationMessages = {
   email: [
    { type: 'required', message: 'Informe o e-Mail' },
    { type: 'pattern', message: 'Informe um email válido' }
   ],
   password: [
    { type: 'required', message: 'Informe a senha' },
    { type: 'minlength', message: 'A senha precisa ter ao menos 6
caracteres' }
  ]
 };

  constructor(
    private navCtrl: NavController,
    private autenticacaoService: AutenticacaoService,
    private formBuilder: FormBuilder
  ) {}

  ngOnInit() {
    this.validationsForm = this.formBuilder.group({
      email: new FormControl('', Validators.compose([
        Validators.required,
        Validators.pattern('^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+.[a-zA-Z0-
9-.]+$')
```

```
      ])),
      password: new FormControl('', Validators.compose([
        Validators.minLength(6),
        Validators.required
      ])),
    });
  }

  registrar(value) {
    this.autenticacaoService.registrar(value)
     .then(res => {
       this.errorMessage = '';
       this.successMessage = 'Sua conta foi criada com sucesso. Se
autentique.';
     }, err => {
       this.errorMessage = err.message;
       this.successMessage = '';
     });
  }

  goLoginPage() {
    this.navCtrl.navigateBack('');
  }
}
```

Para finalizar esta implementação, precisamos importar `ReactiveFormsModule` no arquivo `registrar.module.ts`, pois estamos utilizando o `Reactive Forms` na página.

Quando criamos nossa página de registrar usuário, uma nova rota foi inserida em nosso `app-module.ts`, direcionando para nossa nova página. Em nossa página de login, precisamos dar ao usuário a possibilidade de navegar para esta página. Vamos alterar o link que já temos lá, para o código a seguir.

```
<p style="text-align: center">
    Não tem conta ainda? <a (click)="goToRegistrarPage()">Crie uma
conta</a>
</p>
```

Notou a ligação do `click` com o método `goToRegistrarPage()` ? Então já sabe que precisamos implementá-lo em nossa classe `LoginPage`, tal qual podemos ver na sequência.

```
goToRegistrarPage() {
    this.navCtrl.navigateForward('/registrar');
}
```

Teste a aplicação e vá para a criação de usuários. Insira um email e senha válidos e registre. Veja o retorno na página. Vá para o Firebase e veja os usuários registrados. Depois tente acessar a aplicação com o novo usuário. Na sequência, veja uma figura apresentando a página que acabamos de criar.

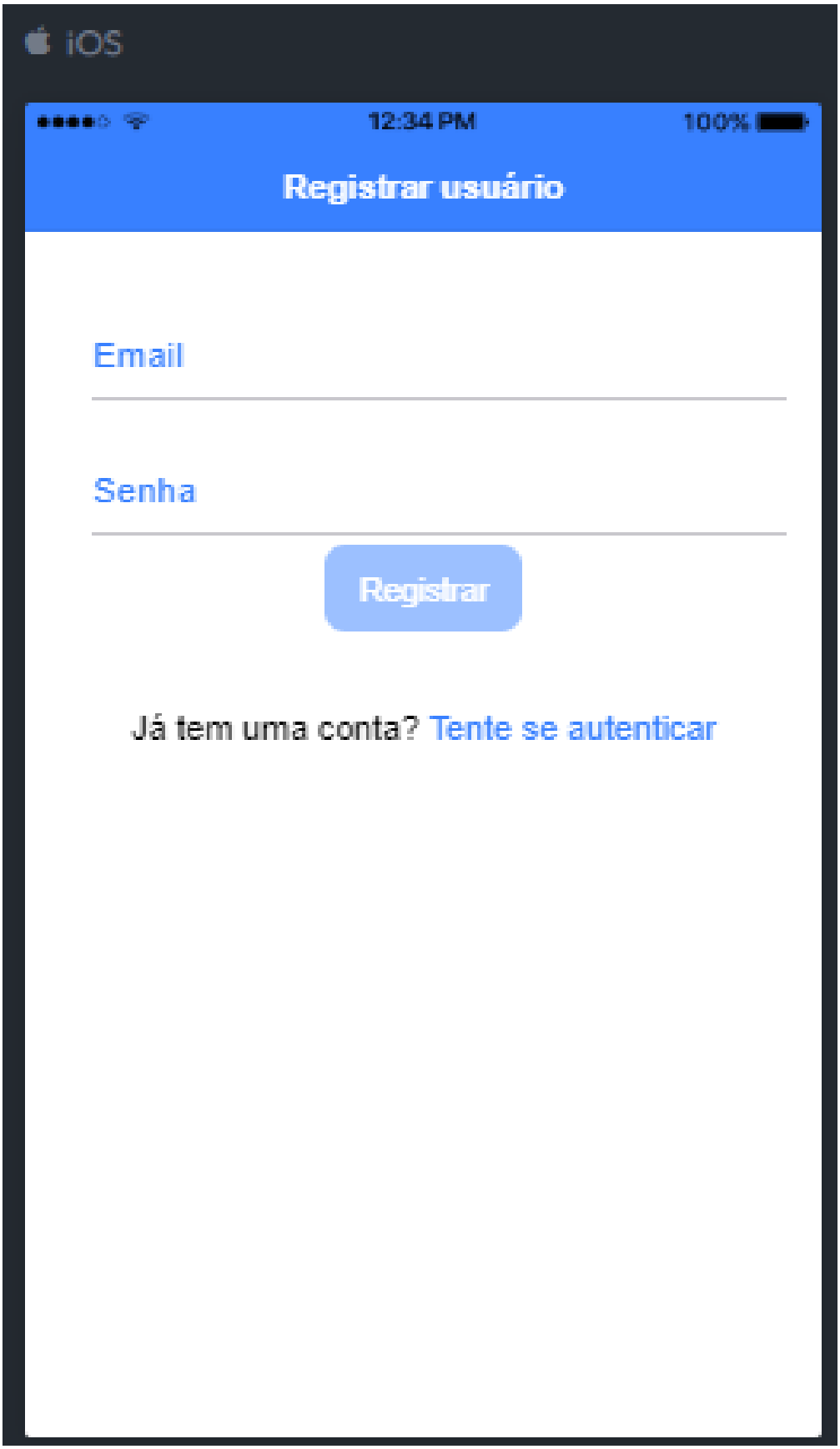

iOS
12:34 PM
100%
Registrar usuário
Email
Senha
Registrar
Já tem uma conta? Tente se autenticar

Figura 9.6: Registro de um novo usuário no Firebase

Conclusão

Fechamos mais um capítulo. Trabalhamos a autenticação de um usuário em nossa aplicação por meio do Firebase Authentication. Possibilitamos o uso de regras que não permitem acesso sem que uma autenticação tenha sido realizada pela aplicação. Utilizamos a API do Firebase também para criarmos um usuário e já o deixarmos apto a acessar nossa aplicação.

Na parte específica do Ionic vimos o `<ion-tabs>` , uma página que possibilita a navegação entre outras por meio de botões na base de uma página principal.

Em relação ao `Authentication` , o Firebase possibilita que configuremos nossa aplicação para que o usuário receba um email após a criação de seu acesso e que o ative apenas por meio deste email. Também está disponível login por outras plataformas, como Google e Facebook, dentre várias outras disponíveis no Console do Firebase. Você pode pensar em trazer isso para sua aplicação. É algo muito comum nos dias atuais e evita que o usuário fique criando contas para todos os aplicativos.

Estamos próximos do término do livro. Teremos só o próximo capítulo para vermos. Trabalharemos o uso de mapas. Será um fechamento bem legal para nosso trabalho.

Capítulo 10
Integração com Google Maps

Neste último capítulo de conteúdo do nosso livro trabalharemos um pouco com mapas, o que se tornou algo comum em aplicações web e para dispositivos móveis e não poderíamos deixar de ver isso aqui. Nós faremos uso da API do Google Maps, com auxílio do plugin `cordova-plugin-googlemaps` e do componente `@ionic-native/google-maps`. Veremos todos os passos para instalação e testes. Executaremos nossa aplicação no browser e em dispositivos Android e iOS. O funcionamento em emuladores para esta funcionalidade não é muito bom e em alguns casos nem chega a funcionar.

Preparando o ambiente para o uso de mapas

O primeiro passo para esta última aplicação será criarmos um novo projeto. Você pode utilizar os projetos anteriores, mas preferi aqui criar um novo. Em seu terminal, na pasta inicial de seus projetos, digite a instrução a seguir.

```
ionic start capitulo10 blank --type=angular --no-git
```

Com nosso projeto criado, começaremos a instalação de plugins e componentes que serão necessários para nossa aplicação. Execute as instruções a seguir, no terminal, na pasta do projeto criado anteriormente.

```
ionic cordova plugin add cordova-plugin-googlemaps

npm install --save @ionic-native/core@5.7.0 @ionic-native/google-maps@5.5.0
```

10.1 Criando nossa aplicação no Google Cloud

Para que possamos fazer uso de qualquer API do Google, como o caso do Google Maps, precisamos realizar um registro no serviço de Cloud do Google. Para isso, acesse o link https://cloud.google.com/maps-platform/#get-started e você terá acesso a uma página com um formulário semelhante ao que se apresenta na figura a seguir. Marque `Maps` e `Places` nele.

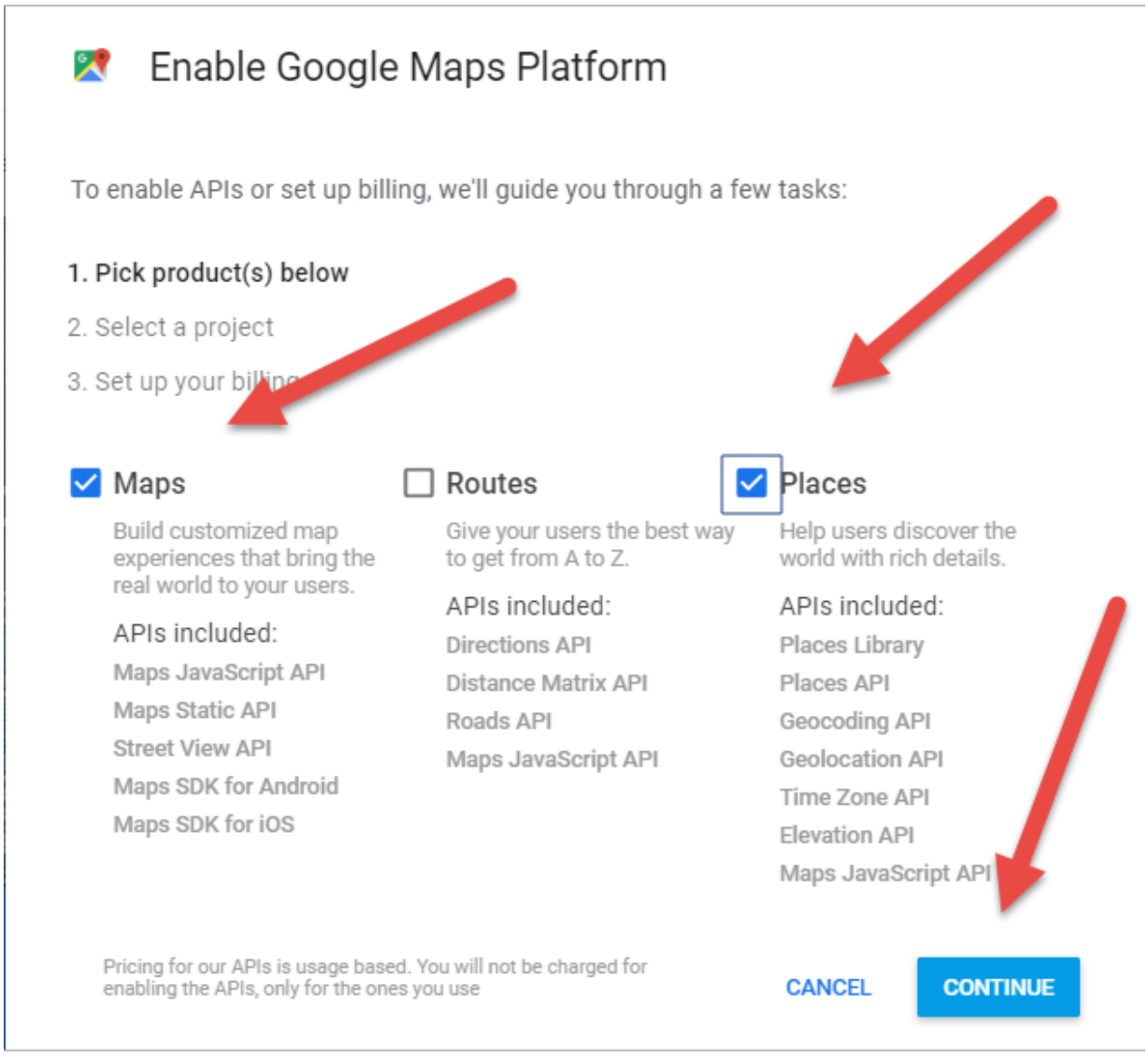

Enable Google Maps Platform
To enable APIs or set up billing, we'll guide you through a few tasks:
1. Pick product(s) below
2. Select a project
3. Set up your billing
Maps
Build customized map experiences that bring the real world to your users.
APIs included:
Maps JavaScript API
Maps Static API
Street View API
Maps SDK for Android
Maps SDK for iOS
Routes
Give your users the best way to get from A to Z.
APIs included:
Directions API
Distance Matrix API
Roads API
Maps JavaScript API
Places
Help users discover the world with rich details.
APIs included:
Places Library
Places API
Geocoding API
Geolocation API
Time Zone API
Elevation API
Maps JavaScript API
Pricing for our APIs is usage based. You will not be charged for enabling the APIs, only for the ones you use
CANCEL
CONTINUE

Figura 10.1: Selecionando a API de mapas do Google

Após esta seleção, uma nova etapa para o formulário será exibida, agora pedindo para selecionarmos um projeto ou criarmos um novo.

Podemos optar pela criação de um novo, caso você ainda não o tenha. Veja a figura a seguir como exemplo.

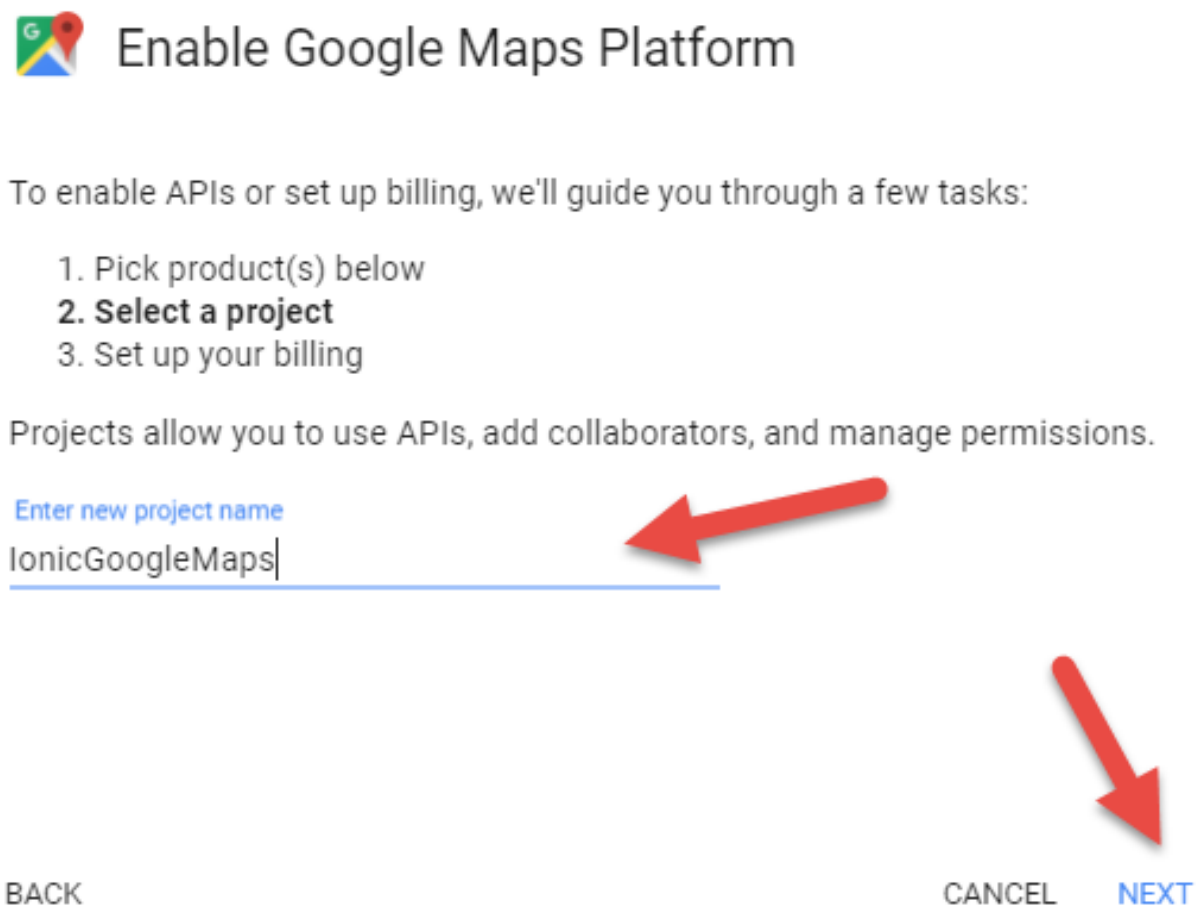

Enable Google Maps Platform
To enable APIs or set up billing, we'll guide you through a few tasks:
1. Pick product(s) below
2. Select a project
3. Set up your billing
Projects allow you to use APIs, add collaborators, and manage permissions.
Enter new project name
IonicGoogleMaps
BACK
CANCEL
NEXT

Figura 10.2: Criando/selecionando a aplicação Google

Até um tempo atrás, o Google não solicitava um cartão de crédito para liberação do uso de algumas APIs. Isso mudou. Precisamos informar um cartão, mas nada será cobrado sem que você autorize. Ou seja, poderemos utilizar esta API aqui sem a preocupação de algo ser debitado. Isso é informado após a confirmação da etapa anterior, com o surgimento da solicitação de confirmação apresentada na figura a seguir.

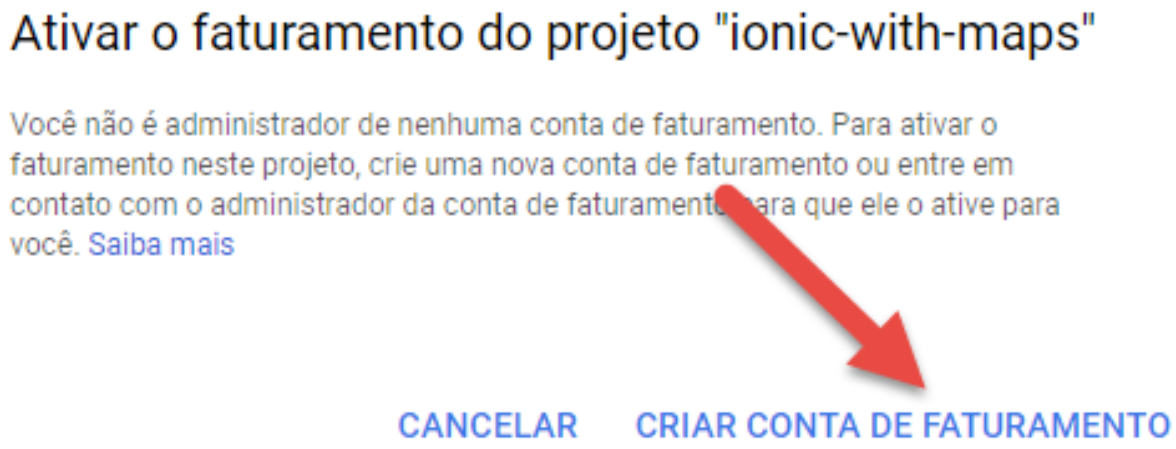

Ativar o faturamento do projeto "ionic-with-maps"
Você não é administrador de nenhuma conta de faturamento. Para ativar o faturamento neste projeto, crie uma nova conta de faturamento ou entre em contato com o administrador da conta de faturamento para que ele o ative para você. Saiba mais
CANCELAR
CRIAR CONTA DE FATURAMENTO

Figura 10.3: Confirmando aceite para informação de cartão

A próxima etapa é a informação de seus dados de contato e cartão de crédito.

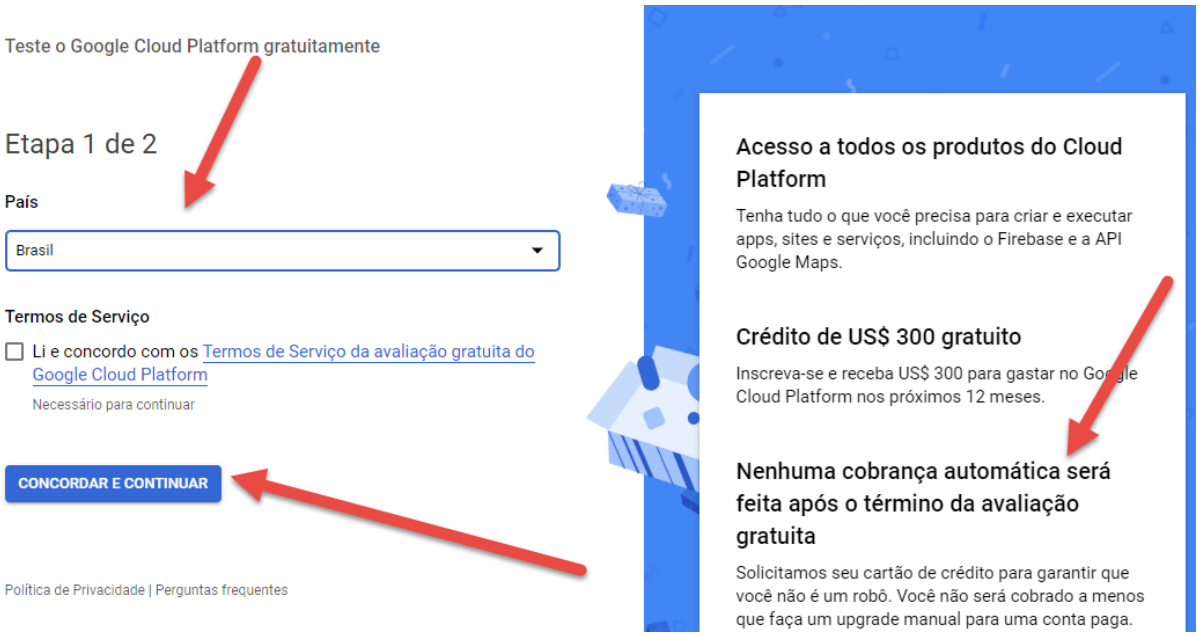

Teste o Google Cloud Platform gratuitamente
Etapa 1 de 2
País
Brasil
Termos de Serviço
Li e concordo com os Termos de Serviço da avaliação gratuita do Google Cloud Platform
Necessário para continuar
CONCORDAR E CONTINUAR
Política de Privacidade | Perguntas frequentes
Acesso a todos os produtos do Cloud Platform
Tenha tudo o que você precisa para criar e executar apps, sites e serviços, incluindo o Firebase e a API Google Maps.
Crédito de US$ 300 gratuito
Inscreva-se e receba US$ 300 para gastar no Google Cloud Platform nos próximos 12 meses.
Nenhuma cobrança automática será feita após o término da avaliação gratuita
Solicitamos seu cartão de crédito para garantir que você não é um robô. Você não será cobrado a menos que faça um upgrade manual para uma conta paga.

Figura 10.4: Informação de dados de contato e cartão

Após confirmar todos seus dados, uma última etapa será visualizada, solicitando confirmação para ativação da plataforma do Google Maps.

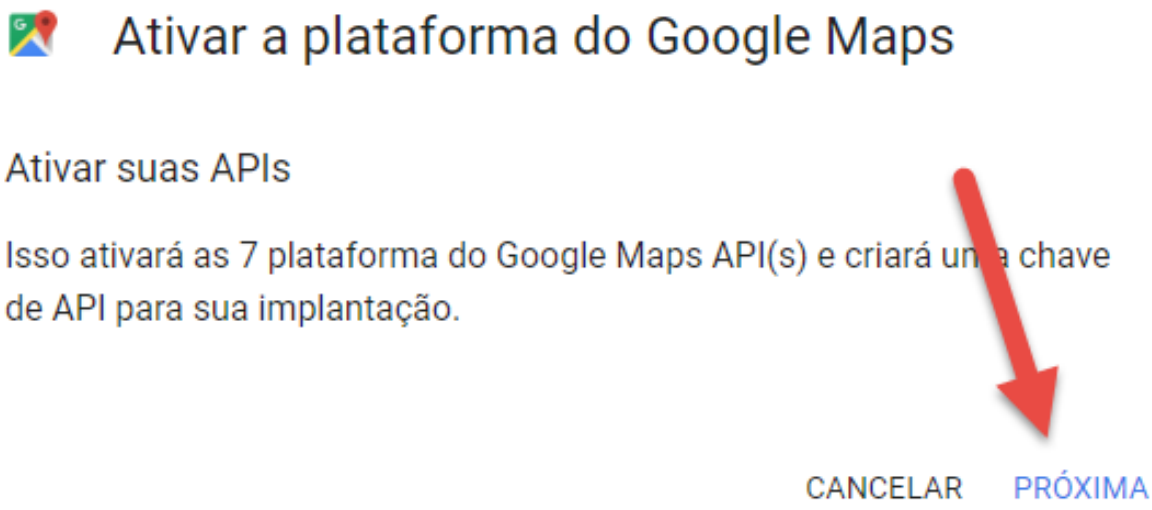

Ativar a plataforma do Google Maps
Ativar suas APIs
Isso ativará as 7 plataforma do Google Maps API(s) e criará uma chave de API para sua implantação.
CANCELAR
PRÓXIMA

Figura 10.5: Ativação da plataforma Google Maps

Para finalizar, será solicitada a você a criação de credenciais para sua aplicação. Após criá-la, copie, pois a utilizaremos em nossas implementações, como já veremos.

10.2 Preparação da aplicação para a visualização de mapas

Nossa aplicação terá uma página de abertura, a `home`, que é criada automaticamente quando criamos nosso projeto. Colocaremos nela uma imagem estática, de apresentação da aplicação e teremos o acesso ao mapa por meio de opções do `Hamburger Menu`.

Já temos então todos os artefatos para a página `home` criados. Vamos agora implementar o que precisamos para essa proposta. Os códigos estão na sequência, em subseções.

home.page.html

```
<ion-header>
  <ion-toolbar>
    <ion-buttons slot="start">
      <ion-menu-button></ion-menu-button>
    </ion-buttons>
    <ion-title>
      Aplicação Google Maps
    </ion-title>
  </ion-toolbar>
</ion-header>

<ion-content>
  <div class="ion-padding">
    <ion-avatar class="main" style="height: 290px;">
        <ion-img src="assets/images/googlemaps.jpg" class="center"></ion-
img>
    </ion-avatar>
  </div>
  <div>
    <ion-card>
      <ion-card-header>
        <ion-card-subtitle>Ionic 4</ion-card-subtitle>
        <ion-card-title>Google Maps Plugin</ion-card-title>
      </ion-card-header>

      <ion-card-content>
        Este aplicativo trará exemplos de utilização da API do Google Maps
por meio do <b>cordova-plugin-googlemaps</b>
        e <b>@ionic-native/google-map</b>. Acesse as opções pelo
<b>Hamburguer menu</b>
      </ion-card-content>
    </ion-card>
  </div>
</ion-content>
```

Veja os componentes do `<ion-header>` no código anterior. Temos o botão para o `Hamburger Menu` e o título da página. Já no `<ion-content>` temos dois `<div>`. O primeiro para a figura, que devemos ter no caminho especificado. Esta figura está dentro de um `<ion-avatar>`, em um `<ion-img>`, e você precisa trazê-la para seu projeto. O `<ion-`

`avatar>` é um componente que formata uma imagem para ser exibida como um círculo, forma usual para algumas aplicações exibirem imagens. A `ion-padding` é oferecida pelo Ionic. No segundo `<div>`, temos um `<ion-card>`.

home.page.scss

```
.main {
    width : 100%;
}

.center {
    margin : 0 auto;
}

ion-avatar ion-img {
    width: 300px !important;
    height: 300px !important;
}
```

As classes que criamos pelo código anterior, `main` e `center`, permitirão a centralização da imagem na página, já a customização do `ion-img`, dentro do `ion-avatar`, define o tamanho da imagem.

home.page.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-home',
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss'],
})
export class HomePage {

}
```

Não precisamos implementar nada no código que renderizará nosso template, mas e o menu que queremos exibir para o usuário acessar o mapa? Vamos implementá-lo em nosso componente `app`.

Também não mexeremos no `home.module.ts`. Vamos então ver os códigos necessários.

app.component.html

```
<ion-split-pane>
  <ion-menu>
    <ion-header>
      <ion-toolbar>
        <ion-title>Menu</ion-title>
      </ion-toolbar>
    </ion-header>
    <ion-content>
      <ion-list>
        <ion-menu-toggle auto-hide="false" *ngFor="let p of paginasApp">
          <ion-item [routerDirection]="'root'" [routerLink]="[p.url]">
            <ion-icon slot="start" [name]="p.icon"></ion-icon>
            <ion-label>
              {{p.titulo}}
            </ion-label>
          </ion-item>
        </ion-menu-toggle>
      </ion-list>
    </ion-content>
  </ion-menu>
  <ion-router-outlet main></ion-router-outlet>
</ion-split-pane>
```

Temos alguns componentes que não tínhamos visto antes, como o `<ion-split-pane>` e o `<ion-menu-toggle>`. O primeiro permite criar um layout chamado de `multi-view`, onde uma página possui dois componentes que interagem entre si. Isso é comum em aplicações para iPad. O segundo é responsável por exibir e ocultar o menu. Dentro dele, temos um `<ion-item>` com informações ligadas à nossa classe, que veremos daqui a pouco. Veja que este componente está dentro de um `<ion-list>` e ele tem, em sua definição, o `*ngFor`. Ao final, antes do fechamento de `<ion-split-pane>`, temos `<ion-router-outlet main></ion-router-outlet>`, que é um componente que integra as rotas com o Angular.

app.component.ts

```
import { Component } from '@angular/core';

import { Platform } from '@ionic/angular';
import { SplashScreen } from '@ionic-native/splash-screen/ngx';
import { StatusBar } from '@ionic-native/status-bar/ngx';
import { Environment } from '@ionic-native/google-maps';

@Component({
  selector: 'app-root',
  templateUrl: 'app.component.html'
})
export class AppComponent {
  public paginasApp = [
    {
      titulo: 'Home',
      url: '/home',
      icon: 'home'
    },
    {
      titulo: 'Minha localização',
      url: '/mylocation',
      icon: 'pin'
    }
  ];

  constructor(
    private platform: Platform,
    private splashScreen: SplashScreen,
    private statusBar: StatusBar
  ) {
    this.initializeApp();
  }

  initializeApp() {
    this.platform.ready().then(() => {
      this.statusBar.styleDefault();
      this.splashScreen.hide();
    });
```

```
  }
}
```

No código anterior, temos a declaração do objeto `paginasApp`. Temos um título, uma URL e um ícone. Estes dados comporão as opções do menu que será exibido pelo HTML que vimos há pouco. Temos dois objetos criados, mas no momento temos apenas a página `home`. A `mylocation` veremos mais adiante. Não mexeremos em `app-routing.module.ts` e nem em `app.module.ts`, pois manteremos o código gerado na criação da aplicação.

10.3 A aplicação em execução

O `<ion-split-pane>` é um componente que possibilita que mais de uma página seja visualizada ao mesmo tempo - já comentamos que isso é comum em iPads, mas também é possível em aplicações executadas no navegador. Então, antes de vermos o aplicativo em um dispositivo, vamos vê-lo no navegador, que é também uma plataforma.

No terminal, na pasta de sua aplicação, vamos instalar a plataforma `browser`, com o código a seguir e, após isso, ver a execução da aplicação. Estamos especificando uma versão mínima para a plataforma, por recomendação do plugin que utilizaremos para os mapas. Após as instruções, temos a figura da aplicação em execução.

```
// Instalação
cordova platform add browser@6.0.0


// Execução
ionic cordova run browser -l
```

Verifique se no navegador está a porta correta, `8100`, pois em meus testes aparecia a `8000` e precisava mudar no navegador. Verifique a

URL também, ela apareceu errado em alguns momentos. A porta correta aparecerá para você no terminal, durante a execução da instrução anterior. Veja as duas visões na figura, menu e página `home`.


Menu
Aplicação Google Maps
Home
Minha localização
Ionic 4
Google Maps Plugin
Este aplicativo trará exemplos de utilização da API do Google Maps por meio do cordova-plugin-googlemaps e @ionic-native/google-map.
Acesse as opções pelo Hamburguer menu

Figura 10.6: Aplicação com menu em execução no navegador

Vamos agora testar nossa aplicação no Android. Veja o código a seguir, seguindo a mesma explicação para o `browser`.

```
// Instalação
cordova platform add android@8.0.0


// Execução
ionic cordova run android -l
```

Ainda que não estejamos utilizando de maneira explícita os componentes plugins para o Google Maps, precisamos registrar em nossa aplicação a API KEY que geramos anteriormente. Isso é feito no arquivo `config.xml` e pode ser feito antes do fechamento da tag `</widget>`, tal qual é mostrado na sequência. Você sabe que tem que utilizar sua API KEY em `value`, certo?

```
<preference name="GOOGLE_MAPS_ANDROID_API_KEY"
value="BIzaSyBAB20kcGPw6ZnIKawiwLaRbejZ36bM8Xg" />
```

```
<preference name="GOOGLE_MAPS_IOS_API_KEY"
value="BIzaSyBAB20kcGPw6ZnIKawiwLaRbejZ36bM8Xg" />
```

Ainda, para a execução, em meu caso, eu precisei atualizar toda minha plataforma Android para a versão `Pie (9.0) API 28 Revision 6`. A aplicação não funcionava com a versão que eu tinha em minha máquina.

Com a atualização da plataforma Android e execução da instrução anterior, ao executar a aplicação, alguns erros surgiram, mas podemos contorná-los, pois na realidade são mudanças na plataforma e, talvez, quando você estiver lendo este livro, isso já esteja corrigido.

Nossas alterações serão realizadas no arquivo `config.xml` e a primeira é a retirada de uma instrução que especifica a versão mínima do Android para o aplicativo. Sendo assim, localize a instrução a seguir e a retire.

```
<preference name="android-minSdkVersion" value="19" />
```

A segunda alteração é a inserção do código a seguir no `config.xml`, abaixo das plataformas.

```
<edit-config file="app/src/main/AndroidManifest.xml" mode="merge"
target="/manifest/application">
  <application android:usesCleartextTraffic="true" />
</edit-config>
```

Para que o código anterior seja válido, precisamos inserir um `namespace xml` no componente `<widget>` logo no início do arquivo. Segue o que deve ser inserido.

```
xmlns:android="http://schemas.android.com/apk/res/android"
```

De acordo com pesquisas que realizei, estas alterações se fizeram necessárias por requisitos da `android@8.0.0`, que instalamos. Mas, novamente, pode ser que quando você teste, isso já esteja transparente, sem precisarmos inferir na configuração. Na figura a seguir, veja a aplicação em execução no emulador Android.

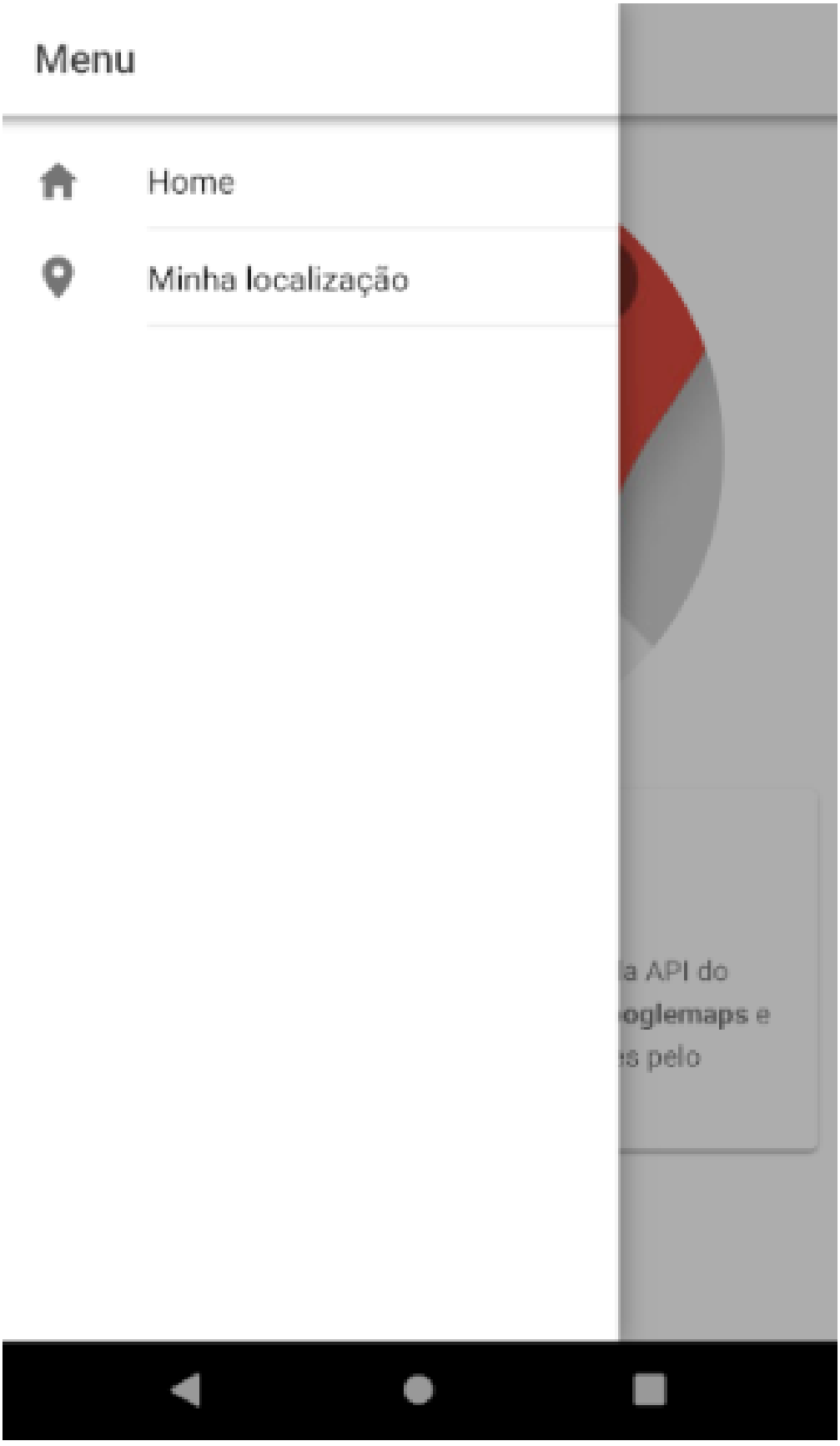

Menu
Home
Minha localização
a API do
oglemaps e
s pelo

Figura 10.7: Aplicação com menu em execução no Android

Agora vamos para o iOS. Veja as instruções a seguir, para instalação e execução da aplicação. Após elas, a figura da aplicação no emulador, com o menu ocultado.

```
// Instalação
cordova platform add ios@5.0.1


// Execução
ionic cordova run ios -l  -- --buildFlag="-UseModernBuildSystem=0"
```

Aplicação Google Maps



IONIC 4

Google Maps Plugin

Este aplicativo trará exemplos de utilização da API do Google Maps por meio do **cordova-plugin-googlemaps** e **@ionic-native/google-map**. Acesse as opções pelo **Hamburguer menu**

Figura 10.8: Aplicação com menu em execução no iOS

10.4 Inserção do mapa na aplicação

Nosso primeiro passo é criar uma página que será renderizada quando o usuário optar por `Minha localização` no menu de opções da aplicação. Sendo assim, no terminal, digite a instrução a seguir.

```
ionic generate page mylocation
```

Essa nossa página exibirá o mapa do Google Maps por meio de nosso plugin, que instalamos no início do capítulo. Vamos então às implementações necessárias para os artefatos, no mesmo molde anterior, para `home` .

mylocation.page.html

```
<ion-header>
    <ion-toolbar>
      <ion-buttons slot="start">
        <ion-menu-button></ion-menu-button>
      </ion-buttons>
      <ion-title>
        Minha localização
      </ion-title>
    </ion-toolbar>
</ion-header>

<ion-content padding>
    <div id="mapDiv">
    </div>
</ion-content>
```

Temos a configuração básica de uma página com a exibição do menu de opções e, na área de conteúdo, um `<div>` , que terá seu conteúdo renderizado por meio de nossa classe para teste template.

mylocation.page.scss

```
#mapDiv {
    width: 100%;
    height: 100%;
    border: 3px solid red;
}
```

Precisamos criar uma classe CSS para nosso `mapDiv`, pois um mapa precisa de pelo menos uma área 100px x 100px para sua renderização. E também vamos aproveitar para outras configurações.

mylocation.page.ts

```
import { Component, OnInit } from '@angular/core';
import { Platform } from '@ionic/angular';
import { GoogleMaps, GoogleMap, Marker, GoogleMapsAnimation, ILatLng }
from '@ionic-native/google-maps';

@Component({
  selector: 'app-mylocation',
  templateUrl: './mylocation.page.html',
  styleUrls: ['./mylocation.page.scss'],
})
export class MylocationPage implements OnInit {

  map: GoogleMap;

  constructor(
    private platform: Platform
  ) { }

  async ngOnInit() {
    await this.platform.ready();
    await this.loadMap();
  }

  async loadMap() {
    const position: ILatLng = {
```

```
      lat: -25.3007931,
      lng: -54.1143221
    };

    this.map = await GoogleMaps.create('mapDiv', {
      camera: {
        target: { position },
        zoom: 18,
        tilt: 30
      }
    });

    const marker: Marker = await this.map.addMarkerSync({
      title: 'UTFPR',
      snippet: 'Melhor universidade de todos os tempos',
      position,
      animation: GoogleMapsAnimation.BOUNCE
    });

    marker.showInfoWindow();
  }
}
```

Antes do construtor, temos a definição de um objeto, chamado `map`, do tipo `GoogleMap`. Ele será nosso mapa. Injetamos um objeto `Platform`, pois precisaremos dele para verificar quando o dispositivo (plataforma) está pronto para o uso, o que nos permitirá invocar o método `loadMap()`, responsável pela renderização do mapa. Neste método, temos a definição de um objeto `ILatLng`, que conterá a latitude e longitude que queremos visualizar no mapa, a posição. Usamos a variável `position` duas vezes no código, para `GoogleMaps.create()` e o `addMarkerSync()`. Atente que estas suas chamadas são `await`, pois elas precisam ser concluídas na ordem em que são especificadas no método.

Em resumo, o método `loadMap()` carregará o Google Maps, centralizará uma determinada posição em nossa página e adicionará uma marca nela. Não entrarei em detalhes sobre os argumentos enviados para os métodos do plugin, é algo extenso e

demanda uma investigação. Recomendo que, caso tenha interesse em se aprofundar, acesse https://github.com/ionic-team/ionic-native-google-maps/blob/master/documents/README.md e realize uma boa leitura.

Precisamos configurar nossa key do Google para execução no navegador de nosso mapa. E isso deve ser feito no `app.component.ts`, no método `initializeApp()`. Veja na sequência.

```
initializeApp() {
  this.platform.ready().then(() => {
      Environment.setEnv({
        // Api key for your server
        // (Make sure the api key should have Website restrictions for
your website domain only)
        API_KEY_FOR_BROWSER_RELEASE:
'BIzaSyBAB20kcGPy6ZnIKawiwLaRbejZ36bM8Xg',

        // Api key for local development
        // (Make sure the api key should have Website restrictions for
'http://localhost' only)
        API_KEY_FOR_BROWSER_DEBUG:
'BIzaSyBAB20kcGPy6ZnIKawiwLaRbejZ36bM8Xg'
      });
      this.statusBar.styleDefault();
      this.splashScreen.hide();
   });
}
```

Veja a aplicação em execução no navegador na figura a seguir.

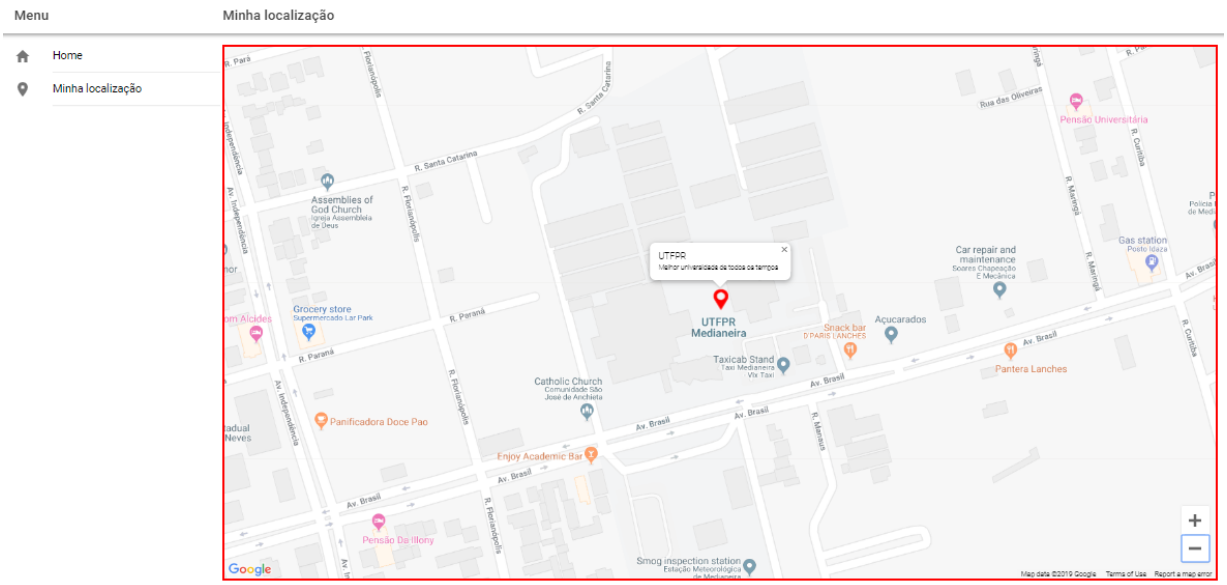


Figura 10.9: Mapa sendo exibido no navegador

Tive um problema ao executar a aplicação neste momento. Ao tentar rodá-la em um dispositivo, o mapa não era exibido e não consegui identificar nenhuma mensagem de erro no console. Depois de algumas tentativas, verifiquei que, estranhamente, para os dispositivos e emuladores, o objeto `position` que criamos não pôde ser utilizado como fizemos para o `browser`, o que me levou a especificar de maneira bruta os valores, tal qual é exibido no código a seguir. Busquei por informações que justificasse este comportamento, mas infelizmente não encontrei, pode ser um bug que possivelmente estará corrigido quando ler o livro.

```
this.map = await GoogleMaps.create('mapDiv', {
  camera: {
    target: {
      lat: -25.3009031,
      lng: -54.1143351
    },
    zoom: 18,
    tilt: 30
  }
});
```

```
const marker: Marker = await this.map.addMarkerSync({
  title: 'UTFPR',
  snippet: 'Melhor universidade de todos os tempos',
  position: {
    lat: -25.3009031,
    lng: -54.1143351
  },
  animation: GoogleMapsAnimation.BOUNCE
});
```

A figura a seguir apresenta a aplicação em execução nas duas plataformas.

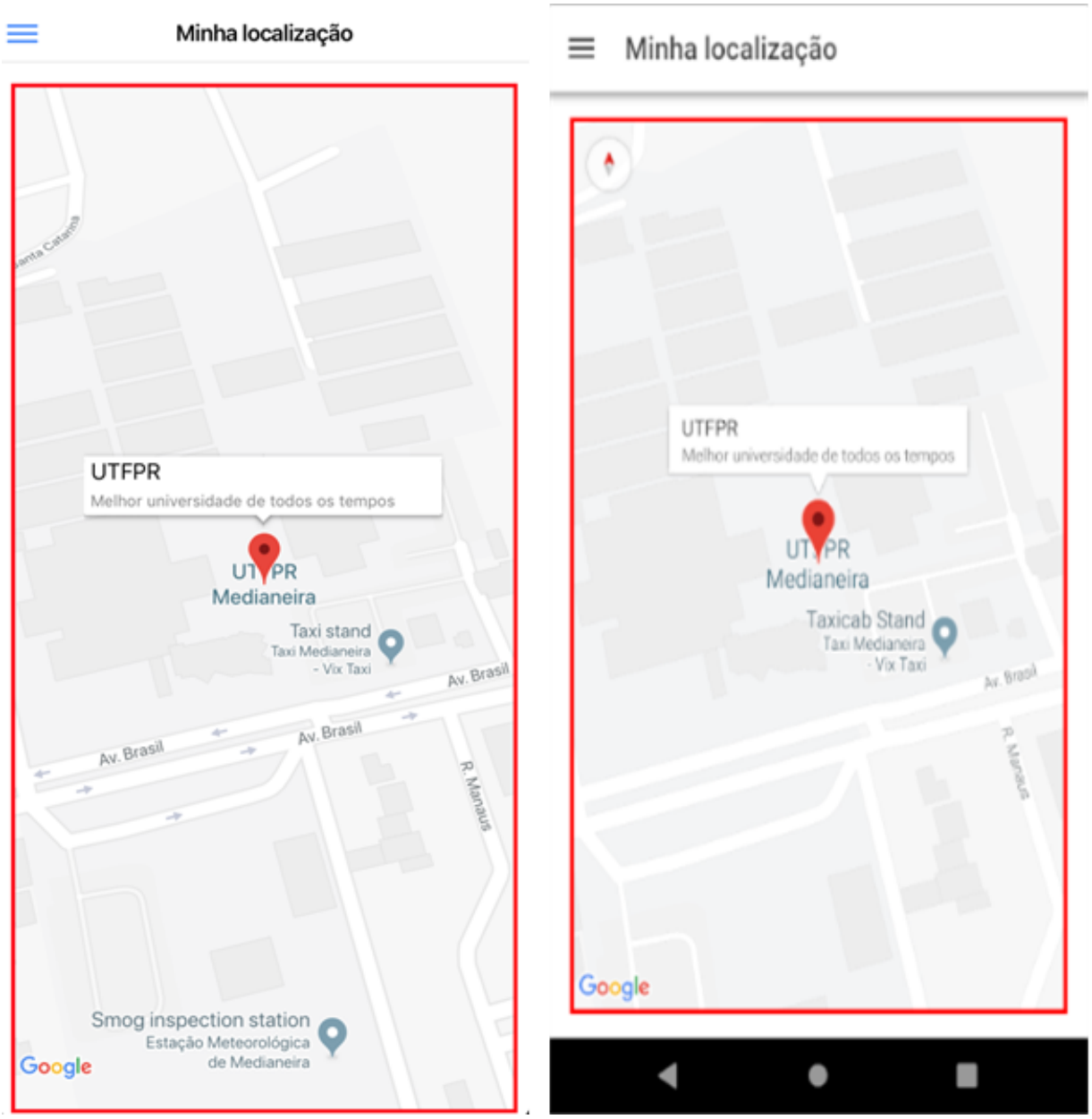


Figura 10.10: Mapa sendo exibido nos emuladores

10.5 Identificação da localização do dispositivo

O mapa exibido pela página criada na seção anterior tinha a localização e exibição de um local específico, o câmpus da UTFPR em Medianeira, local em que trabalho. Mas o nome do componente

é `mylocation`. Deveríamos então pensar em exibir no mapa a nossa localização, ou seja, a localização de nossos dispositivos. Vamos fazer isso, com algumas mudanças em nossos artefatos. A solução não é a mais bonita, mas nosso objetivo é didático.

Nossa primeira alteração será visual, onde inseriremos um botão em nosso mapa que, ao ser pressionado, localizará nossa localização geográfica e a representará em nosso mapa. Para isso, adapte nosso `<div>`, no template HTML para o código a seguir. Note que apenas inserimos um `<ion-button>` com algumas configurações e ligação com um método a ser executado quando for pressionado.

```
<div id="mapDiv">
  <ion-button class="nopadding"  shape="round" color="primary"
size="small" expand="block" (click)="mostrarLocalizacao()">
    Mostre minha localização
  </ion-button>
</div>
```

Agora vamos começar a implementar o necessário em nossa classe `MylocationPage` para que nossa localização possa ser exibida no mapa. A primeira parte está a seguir, logo abaixo da declaração de `map`.

```
loading: any;
```

Quando pressionarmos o botão, precisaremos mostrar para o usuário alguma coisa na tela do dispositivo, para ele não ter a impressão de congelamento da aplicação. Faremos uso do já conhecido `LoadingControler`, que precisamos injetar em nosso construtor, tal qual é mostrado na sequência.

```
private loadingCtrl: LoadingController,
```

Será preciso incluir essa importação do `'@ionic/angular'` no início de nossa classe e, logo após isso, vamos implementar o método ligado com `click` de nosso botão, na página. Veja-o na sequência.

```
async mostrarLocalizacao() {
    // Liberação em relação a qualquer mapa exibido
```

```
    this.map.clear();

    // Criação e exibição da mensagem ao usuário
    this.loading = await this.loadingCtrl.create({
      message: 'Por favor, aguarde...'
    });
    await this.loading.present();

    // Obtenção da localização atual do dispositivo
    this.map.getMyLocation().then((location: MyLocation) => {
      this.loading.dismiss();

      // Efeito visual no movimento do mapa. No browser não aparece.
      this.map.animateCamera({
        target: location.latLng,
        zoom: 18,
        tilt: 30
      });

      // Adição do marcador, que já conhecemos da implementação anterior
      const marker: Marker = this.map.addMarkerSync({
        title: 'Achei você',
        snippet: 'Este é o local aproximado que você está, certo?
Pressione o marcador',
        position: location.latLng,
        animation: GoogleMapsAnimation.BOUNCE
      });

      // Exibição das informações do marcador
      marker.showInfoWindow();

      // Aqui colocamos algo novo, uma mensagem para exibir quando o
usuário clicar no marcador
      marker.on(GoogleMapsEvent.MARKER_CLICK).subscribe(() => {
        // Este método ainda não temos.
        this.showToast('Essa é sua latitude/longitude ' +
location.latLng.toString());
      });
    })
    .catch(err => {
      this.loading.dismiss();
```

```
      this.showToast(err.error_message);
    });
}
```

Em relação ao método `showToast()`, que estamos utilizando no método anterior, faremos uso de nosso também já conhecido `ToastController`, que precisaremos adicionar à importação de `'@ionic/angular'`. Precisamos injetar esta classe em nosso construtor e também implementar o código a seguir.

```
async showToast(message: string) {
    const toast = await this.toastCtrl.create({
      message,
      duration: 4000,
      position: 'middle'
    });

    toast.present();
}
```

O que acha de testarmos nossa aplicação agora nas três plataformas? Note que o botão será exibido no topo do mapa, em azul. Infelizmente (ou felizmente) estes testes precisam ser realizados em dispositivos reais, pois no emulador do Android a aplicação trava e no do iOS a localização não é a real, mas não trava.

10.6 A busca por um endereço

Na seção anterior, iniciamos um mapa com uma localização baseada em Latitude e Longitude, depois, buscamos a localização de nosso dispositivo e vimos que essa nova localização também é exibida na mesma medida. Mas, normalmente, quando buscamos auxílio de um mapa, fazemos uso de um endereço para isso. É o mais usual. Vamos então criar uma nova página. Veja a instrução a seguir, que deve ser inserida em seu terminal.

```
ionic generate page lookforaddress
```

Essa instrução gerará todos os artefatos que precisamos e também a configuração de rotas, mas, em nosso `app.component.ts`, precisamos inserir essa nova opção ao usuário, em nosso objeto que representa as páginas. Vamos inserir lá o código a seguir.

```
,
{
  titulo: 'Localizar endereço',
  url: '/lookforaddress',
  icon: 'navigate'
}
```

Vamos adaptar os arquivos criados e começando com nosso componente HTML, o `lookforaddress.page.html`. Veja-o na sequência, que traz apenas o `<ion-content>`, pois o `<ion-header>` é o mesmo que usamos anteriormente. Estamos utilizando o já conhecido `<ion-grid>`, pois precisaremos de um campo de entrada, `<ion-input>`, para o usuário digitar o endereço. Veja que estamos utilizando `#endereco` para que este componente esteja disponível em nossa classe.

```
<ion-content padding>
  <div id="mapDiv">
    <ion-grid>
      <ion-row>
        <ion-col col-9>
          <ion-input #endereco id="endereco" type="text"
placeholder="Endereço" #endereco></ion-input>
        </ion-col>
        <ion-col col-3>
          <ion-button class="nopadding"  shape="round" color="primary"
size="small" expand="block" (click)="localizarEndereco()">
            Localizar
          </ion-button>
        </ion-col>
      </ion-row>
    </ion-grid>
```

```
  </div>
</ion-content>
```

Precisamos definir características mínimas para nosso `mapDiv`. Em nosso arquivo de estilos, o `scss`, insira as seguintes instruções.

```
#mapDiv {
    width: 100%;
    height: 100%;
    border: 3px solid red;
}
```

Vamos à implementação da classe `LookforaddressPage`, o arquivo `lookforaddress.page.ts`, que será o responsável pela renderização e controle da interação com o usuário. Vejamos os códigos aos poucos.

```
map: GoogleMap;
loading: any;
@ViewChild('endereco') endereco: IonInput;
```

Pela nossa experiência até aqui, consegue identificar onde devemos implementar? Logo após a declaração da classe, pois são objetos que manipularemos em métodos dela. Conhecemos os dois primeiros da implementação anterior e podemos lembrar do `endereco` de outras implementações, onde fazemos a ligação de um componente visual com nossa classe.

Temos também nosso construtor, que é idêntico ao que implementamos no exemplo anterior, mas, para facilitar, trago-o todo na sequência.

```
constructor(
    private platform: Platform,
    public loadingCtrl: LoadingController,
    public toastCtrl: ToastController
) { }
```

Seguindo nossa estrutura, temos o método de inicialização do componente, o `ngOnInit()`, que tem sua implementação a seguir.

Veja que também é semelhante ao que implementamos na seção anterior.

```
async ngOnInit() {
    await this.platform.ready();
    await this.loadMap();
}
```

Notou que estamos chamando `loadMap()` no código anterior? Só que agora, nosso método será diferente, simplificado, pois visualizaremos o mapa apenas após a informação do endereço. Veja esta nova implementação na sequência.

```
async loadMap() {
    this.map = await GoogleMaps.create('mapDiv');
}
```

E o método `localizarEndereco()`, ligado ao nosso botão na interface com o usuário? É nele que constará nossa lógica e instruções para localização de um endereço. Veja-o na sequência. Verifique a invocação de `geocode()`. Ele é quem utilizará, em `address`, o valor informado pelo usuário na página e, caso haja resultado, o mapa então será renderizado, utilizando como base o marcador configurado com a posição identificada, que estará em Latitude e Longitude.

```
async localizarEndereco() {
    this.map.clear();

    this.loading = await this.loadingCtrl.create({
      message: 'Por favor, aguarde...'
    });
    await this.loading.present();

    Geocoder.geocode({
      address: this.endereco.value
    }).then((results: GeocoderResult[]) => {
      this.loading.dismiss();

      if (results.length > 0) {
```

```
        const marker: Marker = this.map.addMarkerSync({
          position: results[0].position,
          title:  JSON.stringify(results[0].position)
        });
        this.map.animateCamera({
          target: marker.getPosition(),
          zoom: 17
        });

        marker.showInfoWindow();
      } else {
        this.showToast('Não encontrado');
      }
    });
}
```

Notou o uso do `showToast()` no código anterior? Pois bem, precisamos dele aqui.

```
async showToast(message: string) {
    const toast = await this.toastCtrl.create({
      message,
      duration: 4000,
      position: 'middle'
    });

    toast.present();
}
```

Execute a aplicação e escolha no menu a nova opção. Não trarei figuras aqui, por serem semelhantes às que já vimos. A mudança está na inserção de uma caixa de texto, para que o usuário possa informar o endereço. O padrão de digitação é o mesmo para o Google Maps.

10.7 Ícone da aplicação e SplashScreen

Você viu, durante todo o livro, que o ícone da aplicação distribuída e a figura da tela de inicialização da aplicação (SplashScreen) é a padrão do Ícone. Chegou a hora de mudarmos isso.

No VSC, na pasta `resources` de nosso aplicativo, temos dois arquivos: `icon.png` e `splash.png`. São estes os arquivos que precisamos substituir, porém, existem algumas regras. Para a imagem de ícone, ela precisa ser no mínimo do tamanho 1024x1024 e a do splash screen de 2732x2732.

Com a substituição das figuras, precisamos, em nosso terminal entrar com a instrução a seguir e informar nossos dados de autenticação. Lembre-se de que é preciso ter a conta gratuita criada no Ionic, mas fizemos isso logo no início do livro.

```
ionic login
```

Para finalizar, com o login realizado, vamos executar as instruções a seguir, para atualização das imagens. Alterações também serão realizadas no `config.xml`.

```
ionic cordova resources ios
ionic cordova resources android
```

Agora é só executar sua aplicação e verificar o resultado.

Conclusão

Terminamos o livro. Neste nosso último contato de aprendizado com o Ionic vimos algumas possibilidades de trabalho com mapas. Visualizamos um mapa básico, buscamos a localização de nosso dispositivo e ainda procuramos por um determinado endereço. Não chegamos nem perto de esgotar o assunto. Existe muita possibilidade e recursos nesta área. Caso você tenha interesse, pode ver muitos recursos em execução que você pode estudar e aplicar aqui: https://docs.google.com/presentation/d/e/2PACX-1vScoho1ensbR4qCI9AIuQN55BZVvK73pAjI7sumDvW3CrxxHnrmp

XWUjx2-8CpFibqU1EjLKCRhuthJ/pub?start=false&loop=false&delayms=3000&slide=id.p.

Fechando o capítulo, trouxemos a possibilidade de alterar nosso ícone da aplicação quando instalada em dispositivos e emuladores, assim como trocar a figura apresentada quando a aplicação é executada, a conhecida SplashScreen.

Tivemos uma boa experiência no estudo deste framework. Foi possível aprendermos muito juntos. Siga em seus estudos. Procure aprofundar seus conhecimentos nos apresentados no livro, descubra novos recursos e componentes que não pudemos ver aqui. E lembre-se de se divertir, sempre.

Capítulo 11
Os estudos não param por aqui

Os dispositivos móveis já fazem parte do dia a dia da maioria da população. Você, como programador(a) ou desenvolvedor(a) não pode perder essa onda.

São duas as maiores plataformas móveis atualmente disponíveis (iOS e Android) e várias são as ferramentas para desenvolvimento de aplicações para elas, quer seja de forma nativa ou híbrida. Neste livro, você teve acesso à metodologia de implementação de aplicações que podem ser executadas nas plataformas descritas, por meio do Ionic, utilizando HTML e TypeScript.

Fizemos um passeio sobre grande parte dos controles disponibilizados pelo Ionic. Também testamos as aplicações em emuladores e em dispositivos físicos, o que nos possibilitou uso de recursos como câmera e álbum de fotografia. Também visualizamos nossas aplicações no navegador e no Electron.

Para a execução nos emuladores e dispositivos fizemos uso do CAPACITOR e do CORDOVA. Frameworks que migram nossas aplicações Ionic para as plataformas específicas e desejadas.

Como o foco da aplicação desenvolvida no livro foi comercial, não podíamos deixar de trabalhar com acesso a base de dados, o que fizemos por meio do SQLite e Firestore. Uma característica comum em aplicações móveis é o sincronismo com aplicações na nuvem e vimos que o Firestore trata isso com maestria.

O livro não esgotou os temas trabalhados. É preciso dedicação para investigação e descoberta de novas tecnologias e recursos. Tenho certeza de que ele foi mais do que um pontapé inicial para o seu desenvolvimento, no que se diz respeito a aplicações móveis.

Espero que, com o conteúdo que trabalhei, tenha sido despertada em você uma grande curiosidade e que o livro tenha desmitificado o desenvolvimento de aplicativos para dispositivos móveis. Agora é bola para a frente na evolução.

Obrigado pela companhia e sucesso.